Professional GEM

by

Tim Oren

PART - I

Windows

IN THE BEGINNING

In GEM, creating a window and displaying it are two different functions. The creation function is called windcreate, and its calling sequence is:

```
handle = windcreate(parts, xfull, yfull, wfull, hfull);
```

This function asks GEM to reserve space in its memory for a new window description, and to return a code or "handle" which you can use to refer to the window in the future. Valid window handles are positive integers; they are not memory pointers.

GEM can run out of window handles. If it does so, the value returned is negative. Your code should always check for this situation and ask the program's user to close some windows and retry if possible. Handle zero is special. It refers to the "desktop", which is predefined as light green (or gray) on the ST. Window zero is always present and may be used, but never deleted, by the programmer.

The xfull, yfull, wfull, and hfull parameters are integers which determine the maximum size of the window. Xfull and yfull define the upper left corner of the window, and wfull and hfull specify its width and height. (Note that all of the window coordinates which we use are in pixel units.)

GEM saves these values so that the program can get them later when processing FULL requests. Usually the best maximum size for a window is the entire desktop area, excepting the menu bar. You can find this by asking windget for the working area of the desktop (handle zero, remember):

```
windget(0, WFWXYWH, &xfull, &yfull, &wfull, &hfull);
```

Note that WFWXYWH, and all of the other mnemonics used in this article, are defined in the GEMDEFS.H file in the ST Toolkit.

The parts parameter of windcreate defines what features will be included in the window when it is drawn. It is a word of single bit flags which indicate the presence/absence of each feature. To request multiple features, the flags are "or-ed" together. The flags' mnemonics and meanings are:

NAME - A one character high title bar at the top of the window.

INFO - A second character line below the NAME.

MOVER - This lets the user move the window around by "dragging" in the NAME area. NAME also needs to be defined.

CLOSER - A square box at the upper left. Clicking this control point asks that the window be removed from the screen.

FULLER - A diamond at upper right. Clicking this control point requests that the window grow to its maximum size, or shrink back down if it is already big.

SIZER - An arrow at bottom right. Dragging the SIZER lets the user choose a new size for the window.

VSLIDE - defines a right-hand scroll box and bar for the window. By dragging the scroll bar, the user requests that the window's "viewport" into the information be moved. Clicking on the gray box above the bar requests that the window be moved up one "page". Clicking below the bar requests a down page movement. You have to define what constitutes a page or line in the context of your application.

UPARROW - An arrow above the right scroll bar. Clicking here requests that the window be moved up one "line". Sliders and arrows almost always appear together.

DNARROW - An arrow below the right scroll bar. Requests that window be moved down a line.

HSLIDE - These features are the horizontal equivalent of the RTARROW above. They appear at the bottom of the window. Arrows LFARROW usually indicate "character" sized movement left and right. "Page" sized movement has to be defined by each application.

It is important to understand the correspondence between window features and event messages which are sent to the application by the GEM window manager. If a feature is not included in a window's creation, the user cannot perform the corresponding action, and your application will never receive the matching message type. For example, a window without a MOVER may not be dragged by the user, and your app will never get a WMMOVED message for that window.

Another important principle is that the application itself is responsible for implementing the user's window action request when a message is received. This gives the application a chance to accept, modify, or reject the user's request.

As an example, if a WMMOVED message is received, it indicates that the user has dragged the window. You might want to byte or word align the requested position before proceeding to move the window. The windset calls used to perform the actual movements will be described in the next article.

OPEN SESAME!

The windopen call is used to actually make the window appear on the screen. It animates a "zoom box" on the screen and then draws in the window's frame. The calling sequence is:

```
windopen(handle, x, y, w, h);
```

The handle is the one returned by windcreate. Parameters x, y, w, and h define the initial location and size of the window. Note that these measurements INCLUDE all of the window frame parts which you have requested. To find out the size of the area inside the frame, you can use

```
windget(handle, WFWXYWH, &innerx, &innery, &innerw, &innerh);
```

Whatever size you choose for the window display, it cannot be any larger than the full size declared in windcreate.

Here is a good place to take note of a useful utility for calculating window sizes. If you know the "parts list" for a window, and its inner or outer size, you can find the other size with the windcalc call:

```
windcalc(parts, kind, inputx, inputy, inputw, inputh, &outputx,
                &outputy, &outputw, &outputh);
```

Kind is set to zero if the input coordinates are the inner area, and you are calculating the outer size. Kind is one if the inputs are the outer size and you want the equivalent inner size. Parts are just the same as in windcreate.

There is one common bug in using windopen. If the NAME feature is specified, then the window title must be initialized BEFORE opening the window:

```
windset(handle, WFNAME, ADDR(title), 0, 0);
```

If you don't do this, you may get gibberish in the NAME area or the system may crash. Likewise, if you have specified the INFO feature, you must make a windset call for WFINFO before opening the window.

Note that ADDR() specifies the 32-bit address of title. This expression is portable to other (Intel-based) GEM systems. If you don't care about portability, then &title[0], or just title alone will work fine on the ST.

CLEANING UP

When you are done with a window, it should be closed and deleted. The call

```
windclose(handle);
```

takes the window off the screen, redraws the desktop underneath it, and animates a "zoom down" box. It doesn't delete the window's definition, so you can reopen it later.

Deleting the window removes its definition from the system, and makes that handle available for reuse. Always close windows before deleting, or you may leave a "dead" picture on the screen. Also be sure to delete all of your windows before ending the program, or your app may "eat" window handles. The syntax for deleting a window is:

```
winddelete(handle);
```

THOSE FAT SLIDERS

One of ST GEM's unique features is the proportional slider bar. Unlike other windowing systems, this type of bar gives visual feedback on the fraction of a document which is being viewed, as well as the position within the document. The catch, of course, is that you have two variables to maintain for each scroll bar: size and position.

Both bar size and position range from 1 to 1000. A bar size of 1000 fills the slide box, and a value of one gets the minimum bar size. To compute the proper size, you can use the formula:

```
size = min(1000, 1000 * seendoc / totaldoc)
```

Seendoc and totaldoc are the visible and total size of the document respectively, in whatever units are appropriate. As an example, if your window could show 20 lines of a 100 line text file, you should set a slider size of 200. Since the window might be bigger than the total document at some points, you need the maximum function. If the document size is zero, force the slider size to 1000. (Note: You will probably need to do

the computation above with 32-bit arithmetic to avoid overflow problems).

Once you have computed the size, use the windset function to configure the scroll bar:

```
windset(handle, WFVSLSIZE, size, 0, 0, 0);
```

This call sets the vertical (right hand) scroll bar. Use WFHSLSIZE for the horizontal scroller. All of these examples are done for the vertical dimension, but the principles are identical in the other direction.

Bar positioning is a little tougher. The most confusing aspect is that the 1-1000 range does not set an absolute position of the bar within the scroll box. Instead, it positions the TOP of the bar within its possible range of variation.

Let's look at our text file example again to make this clearer. If there are always 20 lines of a 100 line file visible, then the top of the window must be always be somewhere between line 1 and line 81. This 80 line range is the actual freedom of movement of the window. So, if the window were actually positioned with its top at line 61, it would be at the three-quarter position within the range, and we should set a scroll bar position of 750. The actual formula for computing the position is:

```
pos = 1000 * (topwind - topdoc) / (totaldoc - seendoc)
```

Topwind and topdoc are the top line in the current window and the whole document, respectively. Obviously, if seendoc is greater or equal to totaldoc, you need to force a zero value for pos. This calculation may seem rather convoluted the first time through, but is easy once you have done it. When you have computed the position, windset configures the scroll bar:

```
windset(handle, WFVSLIDE, pos, 0, 0, 0);
```

WFHSLIDE is the equivalent for horizontal scrolling.

It is a good practice to avoid setting the slider size or position if they are already at the value which you need. This avoids an annoying redraw flash on the screen when it is not necessary. You can check on the current value of a slider parameter with windget:

```
windget(handle, WFVSLIDE, &currvalue, &foo, &foo, &foo);
```

Foo is a dummy variable which needs to be there, but is not

used. Substitute WFVSLIDE with whatever parameter you are checking.

One philosophical note on the use of sliders: It is probably best to avoid the use of both sliders at once unless it is clearly appropriate to the type of data which is being viewed.

Since Write and Paint programs make use of the sheet-of-paper metaphor, moving the window around in both dimensions is reasonable. However, if the data is more randomly organized, such as a tableau of icons, then it is probably better to only scroll in the vertical dimension and "reshuffle" if the window's width is changed. Then the user only needs to manipulate one control to find information which is off-screen. Anyone who has had trouble finding a file or folder within a Desktop window will recognize this problem.

COMING UP NEXT

In my next column in Antic Online, we'll conclude the tour of the ST's windowing system. I'll discuss the correct way to redraw a window's contents, and how to handle the various messages which an application receives from the window manager. Finally, we'll look at a way to redesign the desktop background to your own specifications.

FEEDBACK

One of the beauties of an on-line column is that you can make your comments known immediately. To register your opinions, select ST FEEDBACK, enter your message, leave your name, and enter a blank line to exit.

I am interested in hearing proposals for topics, feedback on the technical level of the column, and reports on bugs and other "features" in both the column and the ST itself. Your comments will be read by the ANTIC staff and myself and, though we might not answer individual questions, they will be used to steer the course of future columns.

PART - II

Windows

EXCELSIOR

In this installment, we continue the exploration of GEM's window manager by finding out how to process the messages received by an application when it has a window defined on the screen.

Also, beginning with this column, sample C code demonstrating the techniques discussed will be available on SIG*ATARI in DL5. This will allow you to download the code without interference by the CIS text-formatter used by ANTIC ONLINE output.

The file for this column is GEMCL2.XMO. All references to non-GEM routines in this column refer to this file. Please note that these files will not contain entire programs. Instead, they consist of small pieces of utility code which you may copy and modify in your own programs.

REDRAWING WINDOWS

One of the most misunderstood parts of GEM is the correct method for drawing within a window. Most requests for redrawing are generated by the GEM system, and arrive as messages (read with evntmulti) which contain the handle of the window, and the screen rectangle which is "dirty" and needs to be redrawn.

Screen areas may become dirty as a result of windows being closed, sized down, or moved, thus "exposing" an area underneath. The completion of a dialog, or closing of a desk accessory may also free up a screen area which needs to be redrawn. When GEM detects the presence of a dirty rectangle, it checks its list of open windows, and sends the application a redraw message for each of its windows which intersects the dirty area.

CAVEAT EMPTOR

GEM does not "clip" the rectangle which it sends to the application; that is, the rectangle may not lie entirely within the portion of the window which is exposed on the screen. It is the job of the application to determine in what portion of the rectangle it may safely draw. This is done by examining the "rectangle list" associated with the window.

A rectangle list is maintained by GEM for each active window. It contains the portions of the window's interior which are exposed, i.e., topmost, on the screen and within which the app may draw.

Let's consider an example to make this clear. Suppose an app has opened two windows, and there are no desk accessory windows open. The window which is topmost will always have only one rectangle in its list. If the two are separate on the screen, then the second window will also have one rectangle. If they overlap, then the top window will "break" the rectangle of the bottom one. If the overlap is at a corner, two rectangles will be generated for the bottom window. If the overlap is on a side only, then three rectangles are required to cover the exposed portion of the bottom window. Finally, if the first window is entirely within the second, it requires four rectangles in the list to tile the second window.

Try working out a few rectangle examples with pencil and paper to get the feel of it. You will see that the possible combinations with more than two windows are enormous. This, by the way, is the reason that GEM does not send one message for each rectangle on the list: With multiple windows, the number of messages generated would quickly fill up the application's message queue.

Finally, note that every app MUST use this method, even if it only uses a single window, because there may be desk accessories with their own windows in the system at the same time. If you do not use the rectangle lists, you may overwrite an accessory's window.

INTO THE BITS

First, we should note that the message type for a redraw request is WMREDRAW, which is stored in msg[0], the first location of the message returned by evntmulti. The window handle is stored in msg[3]. These locations are the same for all of the message types being discuss. The rectangle which needs to be redrawn is stored in msg[4] through msg[7].

Now let's examine the sample redraw code in more detail. The redraw loop is bracketed with mouse off and mouse on calls. If you forget to do this, the mouse pointer will be over-written if it is within the window and the next movement of the mouse will leave a rectangular blotch on the screen as a piece of the "old" screen is incorrectly restored.

The other necessary step is to set the window update flag. This prevents the menu manager from dropping a menu on top of

the screen portion being redrawn. You must release this flag at the end of the redraw, or the you will be unable to use any menus afterwards.

The window rectangles are retrieved using a get-first, get-next scheme which will be familiar if you have used the GEM DOS or PC-DOS wildcard file calls. The end of the rectangle list has been reached when both the width and height returned are zero. Since some part of a window might be off-screen (unless you have clamped its position - see below), the retrieved rectangle is intersected with the desktop's area, and then with the screen area for which a redraw was requested.

Now you have the particular area of the screen in which it is legal to draw. Unless there is only one window in your application, you will have to test the handle in the redraw request to figure out what to put in the rectangle.

Depending on the app, you may be drawing an AES object tree, or executing VDI calls, or some combination of the two. In the AES case, the computed rectangle is used to specify the bounds of the objcdraw. For VDI work, the rectangle is used to set the clipping area before executing the VDI calls.

A SMALL CONFESSION

At the beginning of this discussion, I deliberately omitted one class of redraws: those initiated by the application itself. In some cases a part of the screen must be redrawn immediately to give feedback to the user following a keystroke, button, or mouse action. In these cases, the application could call doredraw directly, without waiting for a message.

The only time you can bypass doredraw, and draw without walking the rectangle list, is when you can be sure that the target window is on top, and that the figure being drawn is entirely contained within it.

In many cases, however, an application initiated redraw happens because of a computed change, for instance, a spreadsheet update, and its timing is not crucial. In this instance, you may wish to have the app send ITSELF a redraw request.

The main advantage of this approach is that the AES is smart enough to see if there is already a redraw request for the same window in the queue, and, if so, to merge the requests by doing a union of their rectangles. In this fashion, the "blinky" appearance of multiple redraws is avoided, without the need to include logic for merging redraws within the program. A

utility routine for sending the "self-redraw" is included in the down-load for this article.

WINDOW CONTROL REQUESTS

An application is notified by the AES, via the message system, when the user manipulates one of the window control points. Remember that you must have specified each control point when the window was created, or will not receive the associated control message.

The most important thing to understand about window control is that the change which the user requested does not take place until the application forwards it to the AES. While this makes for a little extra work, it gives the program a chance to intervene and validate or modify the request to suit.

A second thing to keep in mind is that not all window updates cause a redraw request to be generated for the window, because the AES attempts to save time with raster moves on the screen. Now let's look at each window control request in detail. The message code for a window move is WMMOVED. If you are willing to accept any such request, just do:

```
windset(wh, WFCXYWH, msg[4], msg[5], msg[6], msg[7]);
```

(Remember that wh, the window handle, is always in msg[3]).

The AES will not request a redraw of the window following this call, unless the window is being moved from a location which is partially "off-screen". Instead, it will do a "blit" (raster copy) of the window and its contents to the new location without intervention by the app.

There are two constraints which you may often wish to apply to the user's move request. The first is to force the new location to lie entirely within the desktop, rather than partially off-screen. You can do this with the rcconstrain utility by executing:

```
rcconstrain(&full, &msg[4]);
```

before making the windset call. (Full is assumed to contain the desktop dimensions.)

The second common constraint is to "snap" the x-dimension location of the new location to a word boundary. This operation will speed up GEM's "blit" because no shifting or masking will need to be done when moving the window. To perform this operation, use align() before the windset call:

```
msg[4] = align(msg[4], 16);
```

The message code for a window size request is WMSIZED. Again, if you are willing to accept any request, you can just "turn it around" with the same windset call as given for WMMOVED.

Actually, GEM enforces a couple of constraints on sizing. First, the window may not be sized off screen. Second, there is a minimum window size which is dependent on the window components specified when it was created. This prevents features like scroll arrows from being squeezed into oblivion. The most common application constraint on sizing is to snap the size to horizontal words (as above) and/or vertical character lines. In the latter case, the vertical dimension of the output font is used with align().

Also, be aware that the size message which you receive specifies the EXTERNAL dimensions of the window. To assure an "even" size for the INTERNAL dimensions, you must make a windcalc call to compute them, use align() on the computed values, back out the corresponding external dimensions with the reverse windcalc, and then make the windset call with this set of values.

A window resize will only cause a redraw request for the window if the size is being increased in at least one dimension. This is satisfactory for most applications, but if you must "reshuffle" the window after a size-down, you should send yourself a redraw (as described above) after you make the windset call. This will guarantee that the display is updated correctly. Also note that the sizing or movement of one window may cause redraw requests to be generated for other windows which are uncovered by the change.

The window full request, with code WMFULLED, is actually a toggle. If the window is already at its full size (as specified in the windcreate), then this is a request to shrink to its previous size. If the window is currently small, then the request is to grow to full size.

Since the AES records the current, previous, and maximum window size, you can use windget calls to determine which situation pertains. The hndlfull utility in the down-load (modified from Doodle), shows how to do this.

The "zoom box" effects when changing size are optional, and can be removed to speed things up. Again, if the window's size is decreasing, no redraw is generated, so you must send yourself one if necessary. You should not have to perform any constraint or "snap" operations here, since (presumably) the full and previous sizes have had these checks applied to them

already.

The WMCLOSED message is received when the close box is clicked. What action you perform depends on the application. If you want to remove the window, use windclose as described in the last column. In many applications, however, the close message may indicate that a file is to be saved, or a directory or editing level is to be closed. In these cases, the message is used to trigger this action before or instead of the windclose. (Folders on the Desktop are an example of this situation.)

The WMTOPPED message indicates that the AES wants to bring the indicated window to the "top" and make it active. This happens if the user clicks within a window which is not on top, or if the currently topped window is closed by its application or desk accessory. Normally, the application should respond to this message with:

```
windset(wh, WFTOP, 0, 0);
```

and allow the process to complete.

In a few instances, a window may be used in an output only mode, such as a status display, with at least one other window present for input. In this case, a WMTOPPED message for the status window may be ignored. In all other cases, you must handle the WMTOPPED message even if your application has only one window: Invocation of a desk accessory could always place another window on top. If you fail to do so, subsequent redraws for your window may not be processed correctly.

WINDOW SLIDER MESSAGES

If you specify all of the slider bar parts for your window, you may receive up to five different message types for each of the two sets of sliders. To simplify things a little, I will discuss everything in terms of the vertical (right hand side) sliders. If you are also using the horizontal sliders, the same techniques will work, just use the alternate mnemonics.

The WMVSLID message indicates that the user has dragged the slider bar within its box, indicating a new relative position within the document. Along with the window handle, this message includes the relative position between 1 and 1000 in msg[4].

Recall from last column's discussion that this interval corresponds to the "freedom of movement" of the slider. If you want to accept the user's request, just make the call:

```
windset(wh, WFVSLIDE, msg[4], 0, 0, 0);
```

(Corresponding horizontal mnemonics are WMHSLID and WFHSLIDE).

Note that this windset call will not cause a redraw message to be sent. You must update the display to reflect the new scrolled position, either by executing a redraw directly, or by sending yourself a message.

If the document within the window has some structure, you may not wish to accept all slider positions. Instead you may want to force the scroll position to the nearest text line (for instance). Using terms defined in the last column, you may convert the slider position to "document units" with:

topwind = msg[4] * (totaldoc - seendoc) / 1000 + topdoc

(This will probably require 32-bit arithmetic).

After rounding off or otherwise modifying the request, convert it back to slider units and make the WFVSLIDE request.

The other four slider requests all share one message code: WMARROWED. They are distinguished by sub-codes stored in msg[4]: WAUPPAGE, WADNPAGE, WAUPLINE, and WADNLINE. These are produced by clicking above and below the slider, and on the up and down arrows, respectively. (I have no idea why sub-codes were used in this one instance.) The corresponding horizontal slider codes are: WALFPAGE, WARTPAGE, WALFLINE, and WARTLINE.

What interpretation you give to these requests will depend on the application. In the most common instance, text documents, the customary method is to change the top of window position (topwind) by one line for a WAUPLINE or WADNLINE, and by seendoc (the number of lines in the window) for a WAUPPAGE or WADNPAGE.

After making the change, compute a new slider position, and make the windset call as given above. If the document's length is not an even multiple of "lines" or "pages" you will have to be careful that incrementing or decrementing topwind does not exceed its range of freedom: topdoc to (topdoc + totaldoc - seendoc).

If you have such an odd size document, you will also have to make a decision on whether to violate the line positioning rule so that the slider may be put at its bottom-most position, or to follow the rule but make it impossible to get the slider to the extreme of its range.

A COMMON BUG

It is easy to forget that user clicks are not the only

things that affect slider position. If the window size changes as a result of a WMSIZED or WMFULLED message, the app must also update its sliders (if they are present). This is a good reason to keep the top of window information in "document units".

You can just redo the position calculation with the new "seendoc" value, and call windset. Also remember that changing the size of the underlying document (adding or deleting a bottom line, for instance) must also cause the sliders to be adjusted.

DEPT. OF DIRTY TRICKS

There are two remaining window calls which are useful to advanced programmers. They require techniques which I have not yet discussed, so you may need to file them for future reference.

The AES maintains a quarter-screen sized buffer which is used to save the area under alerts and menu drop-downs. It is occasionally useful for the application to gain access to this buffer for its own use in saving screen areas with raster copies. To do so, use:

```
windget(0, WFSCREEN, &loaddr, &hiaddr, &lolen, &hilen);
```

Hiaddr and loaddr are the top and bottom 16-bits (respectively) of the 32-bit address of the buffer. Hilen and lolen are the two halves of its length.

Due to a preculiarity of the binding you have to reassemble these pieces before using them. (The actual value of WFSCREEN is 17; this does not appear in some versions of the GEMDEFS.H file.)

If you use this buffer, you MUST prevent menus from dropping down by using either the BEGUPDATE or BEGMCTRL windupdate calls. Failure to do so will result in your data being destroyed. Remember to use the matching windupdate: ENDUPDATE or ENDMCTRL, when you are done.

The other useful call enables you to replace the system's desktop definition with a resource of your choosing. The call:

```
windset(0, WFNEWDESK, tree, 0, 0);
```

where tree is the 32-bit address of the object tree, will cause the AES to draw your definition instead of the usual gray or green background. Not only that, it will continue to redraw this tree with no intervention on your part.

Obviously, the new definition must be carefully built to fit the desktop area exactly or garbage will be left around the edges. For the truly sophisticated, a user-defined object could be used in this tree, with the result that your application's code would be entered from the AES whenever the desktop was redrawn. This would allow you to put VDI pictures or complex images onto the desktop background.

A SIN OF OMISSION

In the last column, I neglected to mention that strings whose addresses are passed in the WFNAME and WFINFO windset calls must be allocated in a static data area. Since the AES remembers the addresses (not the characters), a disaster may result if the storage has been reused when the window manager next attempts to draw the window title area.

COMING SOON

This concludes our tour of GEM's basic window management techniques. There have been some unavoidable glimpses of paths not yet taken (forward references), but we will return in time.

On our next excursion, we will take a look at techniques for handling simple dialog boxes, and start exploring the mysteries of resources and object trees.

PART - III

THE DIALOG HANDLER

A MEANINGFUL DIALOG

This issue of ST PRO GEM begins an exploration of ST GEM's dialog handler. I will discuss basic system calls for presenting the dialog, and then continue with techniques for initializing and reading on/off button and "radio" button objects. We will also take some short side-trips into the operation of the GEM Resource Construction Set to assist you in building these dialogs.

There are a number of short C routines which accompany this column. These are stored as file GEMCL3.XMO in DL 5 on SIG*ATARI. Before reading this column, you should visit SIG*ATARI (go pcs-132) and download this file.

DEFINING TERMS

A dialog box is an "interactive form" in which the user may enter text and indicate selections by pointing with the mouse. Dialogs in GEM are "modal", that is, when a dialog is activated other screen functions such as menus and window controls are suspended until the dialog is completed.

In most cases, the visual structure of a GEM dialog is specified within your application's resource file. The GEM Resource Construction Set (RCS) is used to build a picture of the dialog.

When the RCS writes out a resource, it converts that picture into a tree of GEM drawing objects and stores this data structure within the resource. Before your application can display the dialog, it must load this resource file and find the address of the tree which defines the dialog.

To load a resource, the AES checks its size and allocates memory for the load. It then reads in the resource, adjusting internal pointers to reflect the load address. Finally, the object sizes stored in the resource are converted from characters to pixels using the system font size.

A note for those with Macintosh experience: Although Mac and GEM resources share a name, there are fundamental differences which can be misleading. A Mac resource is a fork within a file; a GEM resource is a TOS file by itself. Mac

resources may be paged in and out of memory; GEM resources are monolithic. GEM resources are internally tree structured; Mac resources are not. Finally, Mac resources include font information, while ST GEM does this with font loading at the VDI level.

The resource load is done with the GEM AES call:

```
ok = rsrcload(ADDR("MYAPP.RSC"));
```

"MYAPP" should be replaced with the name of your program. Resources conventionally have the same primary name as their application, with the RSC extent name instead of PRG. The ok flag returned by rsrcload will be FALSE is anything went wrong during the load.

The most common causes of failure are the resource not being in the application's subdirectory, or lack of sufficient memory for GEM to allocate space for the resource. If this happens, you must terminate the program immediately.

Once you have loaded the resource, you find the address of a dialog's object tree with:

```
rsrcgaddr(RTREE, MYDIALOG, &tree);
```

Tree is a 32-bit variable which will receive the address of the root node of the tree.

The mnemonic MYDIALOG should be replaced with the name you gave your dialog when defining it in the RCS. At the same time that it writes the resource, RCS generates a corresponding .H file containing tree and object names. In order to use these mnemonics within your program, you must include the name file in your compile: #include "MYAPP.H"

BUG ALERT!

When using the DRI/Alcyon C compiler, .H files must be in the compiler's home directory or they will not be found. This is especially annoying using a two floppy drive ST development system. The only way around this is to explicitly reference an alternate disk in the #include, for instance: "B:MYAPP.H". [Ed. Note: Use the -i flag with the C pre-processor to name the include directories].

Now that the address of the dialog tree has been found, you are ready to display it. The standard (and minimal) sequence for doing so is given in routine hndldial() in the download. We will now walk through each step in this procedure.

The formcenter call establishes the location of the dialog on the screen. Dialog trees generated by the RCS have an undefined origin (upper-left corner).

Formcenter computes the upper-left location necessary to center the dialog on the screen, and inserts it into the OBX and OBY fields of the ROOT object of the tree. It also computes the screen rectangle which the dialog will occupy on screen and writes its pixel coordinates into variables xdial, ydial, wdial, and hdial.

There is one peculiarity of formcenter which occasionally causes trouble. Normally the rectangle returned in xdial, etc., is exactly the same size as the basic dialog box.

However, when the OUTLINED enhancement has been specified for the box, formcenter adds a three pixel margin to the rectangle returned. This causes the screen area under the outline to be correctly redrawn later (see below). Note that OUTLINED is part of the standard dialog box in the RCS. Other enhancements, such as SHADOWED or "outside" borders are NOT handled in this fashion, and you must compensate for them in your code.

The next part of the sequence is a formdial call with a zero parameter. This reserves the screen for the dialog action about to occur. Note that the C binding given for formdial in the DRI documents is in error: there are nine parameters, not five. The first set of xywh arguments is actually used with formdial calls 1 and 2 only, but place holders must be supplied in all cases.

The succeeding formdial call (parameter one) animates a "zoom box" on the screen which moves and grows from the first screen rectangle given to the second rectangle, where the dialog will be displayed.

The use of this call is entirely optional. In choosing whether to use it or not, you should consider whether the origin of the "zoom" is relevant to the operation. For instance, a zoom from the menu bar is relatively meaningless, while a zoom from an object about to be edited in the dialog provides visual feedback to the user, showing whether the correct object was chosen.

If the origin is not relevant, then the zoom is just a time-waster. If you decide to include these effects, consider a "preferences" option in your app which will allow the experienced and jaded user to turn them off in the interests of speed.

The objcdraw call actually displays the dialog on the screen. Note that the address of the tree, the beginning drawing object, and the drawing depth are passed as arguments, as well as the rectangle allotted for the dialog.

In general, dialogs (and parts of dialogs) are ALWAYS drawn beginning at the ROOT (object zero). When you want to draw only a portion of the dialog, adjust the clipping rectangle, but not the object number. This ensures that the background of the dialog is always drawn correctly.

The objcxywh() utility in the download can be used to find the clipping rectangle for any object within a dialog, though you may have to allow an extra margin is you have used shadows, outlines, or outside borders with the object.

Calling formdo transfers control to the AES, which animates the dialog for user interaction. The address of the dialog tree is passed as a parameter. The second paramter is the number of the editable object at which the text cursor will first be positioned. If you have no text fields, pass a zero. Note that again the DRI documents are in error: passing a -1 default may crash the system. Also be careful that the default which you specify is actually a text field; no error checking is performed.

The formdo call returns the number of the object on which the clicked to terminate the dialog. Usually this is a button type object with the EXIT and SELECTABLE attributes set. Setting the DEFAULT attribute as well will cause an exit on that object is a carriage return is struck while in the dialog.

If the top bit of the return is set, it indicates that the exit object had the TOUCHEXIT attribute and was selected with a double-click. Since very few dialogs use this combination, the sample code simply masks off the top bit.

The next formdial call reverses the "zoom box", moving it from the dialog's location back to the given x,y,w,h. The same cautions apply here as above.

The final formdial call tells GEM that the dialog is complete, and that the screen area occupied by the dialog is now considered "dirty" and needs to be redrawn. Using the methods described in our last column, GEM then sends redraws to all windows which were overlaid, and does any necessary redrawing of the menu or desktop itself.

There is one notable "feature" of formdial(3): It always redraws an area which is two pixels wider and higher than your request! This was probably included to make sure that

drop-shadows were cleaned up, and is usually innocuous.

A HANDY TRICK

Use of the formdial(3) call is not limited to dialogs. You can use it to force the system to redraw any part of the screen. The advantage of this method is that the redraw area need not lie entirely within a window, as was necessary with the sendredraw method detailed in the last column. A disadvantage is that this method is somewhat slower, since the AES has to decide who gets the redraws.

CLEAN UP

As a last step, you need to clear the SELECTED flag in the object which was clicked. If you do not do this, the object will be drawn inverted the next time you call the dialog. You could clear the flag with the GEM objcchange call, but it is inefficient since you do not need to redraw the object.

Instead, use the deselobj() code in the download, which modifies the object's OBSTATE field directly. Assuming that retobj contains the exit object returned by hndldial, the call:

```
deselobj(tree, retobj);
```

will do the trick.

RECAP

The basic dialog handling method I have described contains three steps: initialization (rsrcgaddr), dialog presentation (hndldial), and cleanup (deselobj).

As we build more advanced dialogs, these same basic steps will be performed, but they will grow more complex. The initialization will include setting up proper object text and states, and the cleanup phase will also interrogate the final states of objects to find out what the user did.

BUTTON BUTTON

The simple dialogs described above contain only exit buttons as active objects. As such, they are little more than glorified alert boxes.

We will now increase the complexity a little by considering non-exit buttons. These are constructed by setting the

SELECTABLE attribute on a button object. At run-time, such an object will toggle its state between selected (highlighted) and non-selected whenever the user clicks on it. (You can set the SELECTABLE attribute of other types of objects and use them instead of actual buttons, but be sure that the user will be able to figure out what you intend!)

Having non-exit buttons forces us to consider the problem of initializing them before the dialog, and interrogating and resetting them afterward.

Since a button is a toggle, it is usually associated with a flag variable in the program. As part of the initialization, you should test the flag variable, and if true call:

```
selobj(tree, BTNOBJ);
```

which will cause the button to appear highlighted when the dialog is first drawn. Selobj() is in the download. BTNOBJ is replaced with the name you gave your button when you defined it in the RCS. Since the button starts out deselected, you don't have to do anything if your flag variable is false.

After the dialog has completed, you need to check the object's state. The selectp() utility does so by masking the OBSTATE field. You can simply assign the result of this test to your flag variable, but be sure that the dialog was exited with an OK button, not with a CANCEL! Again, remember to clean up the button with deselobj(). (It's often easiest to deselect all buttons just before you leave the dialog routine, regardless of the final dialog state.)

WHO'S GOT THE BUTTON?

Another common use of buttons in a dialog is to select one of a set of possible options. In GEM, such objects are called radio buttons. This term recalls automobile radio tuners where pushing in one button pops out any others. In like fashion, selecting any one of a set of radio buttons automatically deselects all of the others.

To use the radio button feature, you must do some careful work with the Resource Construction Set.

First, each member of a set of radio buttons must be children of the same parent object within the object tree. To create this structure, put a hollow box type object in the dialog, make it big enough to hold all of the buttons, and then put the buttons into the box one at a time.

By nesting the buttons within the box object, you force them to be its children. Each of the buttons must have both the SELECTABLE and RADIO BUTTON attributes set. When you are done, you may make the containing box invisible by setting its border to zero, but do not FLATTEN it!

Since each radio button represents a different option, you must usually assign a name to each object. When initializing the dialog, you must check which option is currently set, and turn on the corresponding button only. A chain of if-then-else structures assures that only one button will be selected.

At the conclusion of the dialog, you must check each button with selectp() and make the appropriate adjustments to internal variables. Again, an if-then-else chain is appropriate since only one button may be selected. Either deselect the chosen button within this chain or do them all at the end.

There is one common use of radio buttons in which you may short-cut this procedure. If the buttons each represent one possible value of a numeric variable, for instance, a set of selector buttons representing colors from zero to seven, then you can compute the initial object directly.

In order for this technique to work, you must use a special capability of the RCS. Insert the object corresponding to a zero value at the top (or left) of your array of buttons, then put the "one" button below (or right) of it, and so on.

When the buttons are complete, the SORT operation is used to guarantee that the top/left object is in fact the first child of the parent box with the others following in order. Due to the details of object tree structure (to be discussed in the next column), this will guarantee that these objects are contiguous in the resource.

If you assign a name (say BUTTON1) to the first button, then you can initialize the correct button with the call:

```
selobj(tree, BUTTON1 + field);
```

where field is the variable of interest.

When the dialog is complete, you can scan the radio buttons to compute the new value for the underlying variable. The encode() procedure in the download will do this. As always, remember to deselect the buttons at the end.

You can use offsets or multipliers if your variable's values don't start with zero or increment by one. If the values are irregular you may be able to use a lookup table, at the cost

of additional code.

COMING UP NEXT

In the next column, I will discuss the internal structure of object trees. Then we'll use that knowledge to build a piece of code which will "walk" an entire tree and apply a function to each object. We'll apply this code to do all of the button deselects with a single call! I'll also look at handling editable text fields and discuss some ways to alter a dialog's appearance at run-time.

DISPELL GREMLINS

An editing error caused an omission in the first installment of ST PRO GEM. The window components RTARROW and DNARROW should have been listed along with HSLIDE as the horizontal equivalents of the vertical slider components which were discussed.

PART - IV

Resource Structure

Welcome to the fourth installment of ST PRO GEM. We are about to delve into the mysteries of GEM resource structure, and then use this knowledge to create some useful utilities for handling dialogs. As with the past columns, there is once again a download file. You will find it under the name GEMCL4.C in the ATARI 16-bit Forum (GO PCS-58).

The first and largest part of the download contains a C image of a sample resource file. To create this listing, I used the GEM Resource Construction Set to create a dummy resource with three dialogs including examples of all object types, then enabled the C output option and saved the resource. If you have access to a copy of RCS, I suggest that you create your own listing in order to get a feel for the results. Then, using either listing as a roadmap to the resource, you can follow along as we enter...

A MAZE OF TWISTY LITTLE PASSAGES.

While a GEM resource is loaded as a block of binary information, it is actually composed of a number of different data structures. These structures are linked together in a rather tangled hierarchy. Our first job is to map this linkage system.

The topmost structure in a resource file is the resource header. This is an array of words containing the size and offset within the resource of the other structures which follow. This information is used by GEM during the resource load process, and you should never need to access it. (The resource header does not appear in the C output file; it is generated by the RSCREATE utility if the C file is used to recreate the resource.)

The next structure of interest is the tree index. This is an array of long pointers, each of which addresses the beginning of an object tree. Again, you wouldn't normally access this structure directly. The GEM rsrcgaddr call uses it when finding trees' addresses. This structure is called "rstrindex" in the C output.

If you look at the contents of rstrindex you will notice that the values are integers, instead of the pointers I described. What has happened is that RCS has converted the pointers to indices into the object array. (If you actually used the C file to recreate the resource file, then the pointers

would be regenerated by RSCREATE.)

Now you can follow the link from rstrindex to the objects stored in rsobject. Take (for instance) the second entry in rstrindex and count down that many lines in rsobject. The following line (object) should start with a -1. This indicates that it is the root object of a tree. The following objects down to the next root belong to that tree. We'll pass over the details of inter-object linkage for now, leaving it for a later column.

There are a number of different fields in an object, but right now we'll concentrate on two of them: OBTYPE and OBSPEC. The OBTYPE is the field which contains mnemonics like GSTRING and GBOX indicating the type of the object. The OBSPEC is the only field in each object which is a LONG - you can tell it by the L after the number.

What's in OBSPEC depends on the object type, so we need to talk about what kinds of objects are available, what you might use them for, and finally how they use the OBSPEC field.

The box type objects are GBOX, GIBOX, and GBOXCHAR. A GBOX is an opaque rectangle, with an optional border. It's used to create a solid patch of color or pattern on which to place other objects. For instance, the background of a dialog is a GBOX.

A GIBOX is a hollow box which has only a border. (If the border has no thickness, then the box is "invisible", hence the name.) The favorite use for IBOXes is to hold radio buttons. There is also one neat trick you can play with an IBOX. If you have more than one object (say an image and a string) which you would like to have selected all at once, you can insert them in a dialog, then cover them with an IBOX. Since the box is transparent, they will show through. If you now make the box selectable, clicking on it will highlight the whole area at once!

The GBOXCHAR is just like a GBOX, except that a single character is drawn in its center. They are mostly used as "control points": the FULLER, CLOSER, SIZER, and arrows in GEM windows are BOXCHARs, as are the components of the color selection gadgets in the RCS.

The OBSPEC for box type objects is a packed bit array. Its various fields contain the background color and pattern, the border thickness and color, and the optional character and its color.

The string type objects are GSTRING, GBUTTON, and GTITLE. GSTRINGs (in addition to being a bad pun) are for setting up static explanatory text within dialogs. The characters are

always written in the "system font": full size, black, with no special effects.

We have already discussed many of the uses of GBUTTONs. They add a border around the text. The thickness of a GBUTTON's border is determined by what flags are set for the object. All buttons start out with a border thickness of one pixel. One pixel is added if the EXIT attribute is set, and one more is added if the DEFAULT attribute is set.

The GTITLE type is a specially formatted text string used only in the title bar of menus. This type is needed to make sure that the menus redraw correctly. The Resource Construction Set automatically handles inserting GTITLEs, so you will seldom use them directly.

In a resource, the OBSPEC for all string objects is a long pointer to a null terminated ASCII string. The string data in the C file is shown in the BYTE array rsstrings. Again you will notice that the OBSPECs in the C file have been converted to indices into rsstring. To find the string which matches the object, take the value of OBSPEC and count down that many lines in rsstrings. The next line is the correct string.

The formatted text object types are GTEXT, GBOXTEXT, GFTEXT, and GFBOXTEXT. GTEXTs are a lot like strings, except that you can specify a color, different sizes, and a positioning rule for the text. Since they require more memory than GSTRINGs, GTEXTs should be used sparingly to draw attention to important information within a dialog. GTEXTs are also useful for automatic centering of dialog text which is changed at run-time. I will describe this technique in detail later on.

The GBOXTEXT type adds a solid background and border to the GTEXT type. These objects are occasionally used in place of GBUTTONs when their color will draw attention to an important object.

The GFTEXT object is an editable text field. You are able to specify a constant "template" of characters, a validation field for those characters which are to be typed in, and an initial value for the input characters. You may also select color, size, and positioning rule for GFTEXTs. We'll discuss text editing at length below.

The GFBOXTEXT object, as you might suspect, is the same as GFTEXT with the addition of background and border. This type is seldom used: the extra appearance details distract attention from the text being edited.

The OBSPEC for a formatted text object is a pointer to yet another type of structure: a TEDINFO. In the C file, you will

find these in rstedinfo. Take the OBSPEC value from each text type object and count down that many entries in rstedinfo, finding the matching TEDINFO on the next line. Each contains pointers to ASCII strings for the template, validation, and initialization. You can find these strings in rsstrings, just as above.

There are also fields for the optional background and border details, and for the length of the template and text. As we will see when discussing editing, the most important TEDINFO fields are the TEPTEXT pointer to initialized text and the TETXTLEN field which gives its length.

The GIMAGE object type is the only one of its kind. A GIMAGE is a monochrome bit image. For examples, see the images within the various GEM alert boxes. Note that monochrome does not necessarily mean black. The image may be any color, but all parts of it are the SAME color. GIMAGEs are used as visual cues in dialogs. They are seldom used as selectable items because their entire rectangle is inverted when they are clicked. This effect is seldom visually pleasing, particularly if the image is colored.

GIMAGE objects have an OBSPEC which is a pointer to a further structure type: the BITBLK. By now, you should guess that you will find it in the C file in the array rsbitblk. The BITBLK contains fields describing the height and width of the image in pixels, its color,nd it also contains a long pointer to the actual bits which make up the image. In the C file, the images are encoded as hexadecimal words and stored in arrays named IMAG0, IMAG1, and so on.

The last type of object is the GICON. Like the GIMAGE, the GICON is a bit image, but it adds a mask array which selects what portions of the image will be drawn, as well as an explanatory text field. A GICON may also specify different colors for its "foreground" pixels (the ones that are normally black), and its "background" pixels (which are normally white).

The pictures which you see in Desktop windows are GICONs, and so are the disks and trashcan on the desktop surface. With the latter you will notice the effects of the mask. The desktop shows through right up to the edge of the GICON, and only the icon itself (not a rectangle) is inverted when a disk is selected.

The OBSPEC of an icon points to another structure called an ICONBLK. It is shown in the C file as rsiconblk. The ICONBLK contains long pointers to its foreground bit array, to the mask bit array, and to the ASCII string of explanatory text. It also has the foreground and background colors as well as the location of the text area from the upper left of the icon. The most

common use of GICONs and ICONBLKs is not in dialogs, instead they are used frequently in trees which are build at run-time, such as Desktop windows. In a future article, we will return to a discussion of building such "on-the-fly" trees with GICONs.

Now, let's recap the hierarchy of resource structures: The highest level structures are the resource header, and then the tree index. The tree index points to the beginning of each object tree. The objects making up the tree are of several types, and depending on that type, they may contain pointers to ASCII strings, or to TEDINFO, ICONBLK, or BITBLK structures. TEDINFOs contain further pointers to strings; BITBLKs have pointers to bit images; and ICONBLKs have both.

PUTTING IT TO WORK

The most common situations requiring you to understand resource structures involve the use of text and editable text objects in dialogs. We'll look at two such techniques.

Often an application requires two or more dialogs which are very similar except for one or two title lines. In this circumstance, you can save a good deal of resource space by building only one dialog, and changing the title at run time.

It is easy to go wrong with this practice, however, because the obvious tactic of using a GSTRING and writing over its text at run time can go wrong. The first problem is that you must know in advance the longest title to be used, and put a string that long into the resource. If you don't you will damage other objects in the resource as you copy in characters. The other problem is that a GSTRING is always drawn at the same place in a dialog. If the length of the title changes from time to time, the dialog will have an unbalanced and sloppy appearance.

A better way to do this is to exploit the GTEXT object type, and the TEDINFO structure. The settext() routine in the download shows how. The parameters provided are the tree address, the object number, and the 32-bit address of the string to be substituted. For this to work, the object referenced should be defined as a GTEXT type object. Additionally, the Centered text type should be chosen, and the object should have been "stretched" so that it fills the dialog box from side to side.

In the code, the first action is to get the OBSPEC from the object which was referenced. Since we know that the object is a GTEXT, the OBSPEC must point to a TEDINFO. We need to change two fields in the TEDINFO. The TEPTEXT field is the pointer to the actual string to be displayed; we replace it with the address of our new string. The TETXTLEN field is loaded with the

new string's length. Since the Centered attribute was specified for the object, changing the TETXTLEN will cause the string to be correctly positioned in the middle of the dialog!

Editing text also requires working with the TEDINFO structure. One way of doing this is shown in the download. The object to be used (EDITOBJ) is assumed to be a GFTEXT or GFBOXTEXT. Since we will replace the initialized text at run time, that field may be left empty when building the object in the RCS.

The basic trick of this code is to point the TEDINFO's TEPTEXT at a string which is defined in your code's local stack. The advantages of this technique are that you save resource space, save static data by putting the string in reusable stack memory, and automatically create a scratch string which may be discarded if the dialog is cancelled.

The text string shown is arbitrarily 41 characters long. You should give yours a length equal to the number of blanks in the object's template field plus one. Note that the code is shown as a segment, rather than a subroutine. This is required because the text string must be allocated within the context of dialog handling routine itself, rather than a routine which it calls!

After the tree address is found, the code proceeds to find the TEDINFO and modify its TEPTEXT as described above. However, the length which is inserted into TETXTLEN must be the maximum string length, including the null!

The final line of code inserts a null into the first character of the uninitialized string. This will produce an empty editing field when the dialog is displayed. If there is an existing value for the object, you should instead use strcpy() to move it into text[]. Once the dialog is complete, you should check its final status as described in the last article. If an "OK" button was clicked, you will then use strcpy() to move the value in text[] back to its static location.

Although I prefer this method of handling editable text, another method deserves mention also. This procedure allocates a full length text string of blanks when creating the editable object in the RCS. At run-time, the TEPTEXT link is followed to find this string's location in the resource, and any pre-existing value is copied in. After the dialog is run, the resulting value is copied back out if the dialog completed successfully.

Note that in both editing techniques a copy of the current string value is kept within the application's data area.

Threading the resource whenever you need to check a string's value is extremely wasteful.

One final note on editable text objects: GEM's editor uses the commercial at sign '@' as a "meta-character". If it is the first byte of the initialized text, then the field is displayed blank no matter what follows. This can be useful, but is sometimes confusing when a user in all innocence enters an @ and has his text disappear the next time the dialog is drawn!

LETTERS WE GET LETTERS

The Feedback section on ANTIC ST ONLINE is now functional and is producing a gratifying volume of response. A number of requests were made for topics such as ST hardware and ST BASIC which are beyond the intended scope of this column. These have been referred to ANTIC's editorial staff for action.

So many good GEM questions were received that I will devote part of the next column to answering several of general interest. Also, your requests have resulting in scheduling future columns on VDI text output and on the principles (or mythology) of designing GEM application interfaces. Finally, a tip of the hat to the anonymous reader who suggested including the actual definitions of all macro symbols, so that those without the appropriate H files can follow along. As a result of this suggestion, the definitions for this column and the previous three are included at the end of the download. Future articles will continue this practice.

STRAW POLL!

I'd like to make a practice of using the Feedback to get your opinions on the column's format. As a first trial, I'd like to know your feelings about my use of "portability macros" in the sample code. These macros, LLGET for example, are used for compatibility between 68K GEM systems like the ST, and Intel based systems like the IBM PC. This may be important to many developers. On the other hand, omitting them results in more natural looking C code. For instance, in the download you will find a second version of settext() as described above, but without the portability macros. So, I would like to know if you think we should (A) Keep the macros - portability is important to serious developers, (B) Get rid of them - who cares about Intel chips anyway, or (C) Who cares? I'll tally the votes in two weeks and announce the results here.

STAY TUNED!

As well as answers to feedback questions, the next column will discuss how GEM objects are linked to form trees, and how to use AES calls and your own code to manipulate them for fun and profit. In the following installment, we'll look at the VDI raster operations (also known as "blit" functions).

PART - V

Resource Tree Structures

This is the fifth issue of ST PROFESSIONAL GEM, concluding our trek through GEM dialogs and resources with a look at the internal structure of object trees. Also, I'll answer a number of questions of general interest which have been received via the ANTIC ONLINE FEEDBACK. As always, there is a download file associated with this column: GEMCL5.C, which you will find in DL3 of the new Atari 16-bit SIG (type GO PCS-58 or GO ATARI16).

Even if you have no immediate use for this issue's code, be sure to take the download anyway; some of the routines will be used in later articles.

In the last installment, we established that resources trees are pointed to by the tree index, and that they are composed of objects which contain pointers onward to other structures. However, we passed over the issue of linkage among the objects within a tree. It is now time to go back and cure this omission.

The technical term for the linkage scheme of an object tree is a "right-threaded binary tree". If you already know what this is, you can skim over the next few paragraphs. If you happen to have access to a copy of the book "FUNDAMENTAL ALGORITHMS", which is part of the series THE ART OF COMPUTER PROGRAMMING by Donald E. Knuth, you might want to read his excellent discussion of binary trees beginning on page 332.

For the following discussion, you should have a listing of the C image of a resource tree in front of you. For those who do not have the listing from the last column, I have included a fragment at the beginning of the download. Before we begin, I should warn you of one peculiarity of "computer trees": They grow upside-down! That is, when they are diagrammed or described, their root is at the top, and the "leaves" grow downward. You will see this both in the listing, and in the way the following discussion talks about moving through trees.

Each GEM object tree begins at its ROOT object, numbered zero, which is the object pointed at by the tree index. There are three link fields at the beginning of each object. They are called OBNEXT, OBHEAD, and OBTAIL, which is the order in which they appear.

Each of the links is shown as an index relative to the root of the current tree. This means that the link '0' would refer to the root of the tree, while '2' would indicate the object two lines below it. The special link -1 is called NIL, and means

that there is no link in the given direction.

Each object, or node, in a tree may have "offspring" or nodes which are nested below it. If it does, then its OBHEAD will point to its first (or "leftmost") "child", while the OBTAIL will point to the last ("rightmost") of its offspring. The OBNEXT pointer links the children together, with the OBNEXT of the first pointing to the second, and so on, until the OBNEXT of the last finally points back to its parent, the object at which we started.

Remember that each of these children may in turn have offspring of their own, so that the original "parent" may have a large and complex collection of "descendents".

Let's look at the first tree in the download to see an example of this structure. The very first object is the ROOT. Note that its OBNEXT is NIL, meaning that there are no more objects in the tree: the ROOT is both the beginning and the end of the tree. In this case, the OBHEAD is 1 and the OBTAIL is 3, showing that there are at least two different children.

Following OBHEAD down to the next line, we can trace through the OBNEXT links (2, 3, 0) as they lead through a total of three children and back to the ROOT. You will notice that the first two children have NIL for the OBHEAD and OBTAILs, indicating that they have no further offspring.

However, node three, the last child of the ROOT, does have the value 4 for both its OBHEAD and OBTAIL. By this we can tell that it has one, and only one, offspring. Sure enough, when we look at node four, we see that its OBNEXT leads immediately back to node three. Additionally, it has no further offspring because its OBHEAD and OBTAIL are NIL.

You will find that object trees are always written out by the Resource Construction Set in "pre-order". (Again, see Knuth if you have a copy.) This means that the ROOT is always written first, then its offspring left to right. This rule is applied recursively, that is, we go down to the next level and write out each of these nodes, then THEIR children left to right, and so on.

For a further example, look at the next tree in rsobject in the download. You will see that the ROOT has an OBHEAD of 1 and an OBTAIL of 6, but that it actually has only three offspring (nodes 1, 2 and 6). We see that node 2 itself had children, and applying the rule given above, they were written out before continuing with the next child of the ROOT.

Why was this seemingly complex structure chosen for GEM? The reason has do with the tasks of drawing objects in their

proper locations on the screen, and determining which object was "hit" when a mouse click is detected.

To find out how this works, we must look at four more fields found in each object: OBX, OBY, OBWIDTH, and OBHEIGHT. These fields are the last four on each line in the sample trees.

Each object in a tree "owns" a rectangle on the screen. These fields define that rectangle. When a resource is stored "outside" the program the fields are in character units, so that an object with OBWIDTH of 10 and OBHEIGHT of 2 (for instance) would define a screen area 10 characters wide and 2 high.

When the resource is read into memory with an rsrcload call, GEM multiplies the appropriate character dimension in pixels into each of these fields. In this way portability is achieved: the same resource file works for any of the ST's three resolutions. Knowing how rsrcload works, your code should treat these fields as pixel coordinates.

I have committed one oversimplification above. If an object is not created on a character boundary in the RCS, then the external storage method described will not work. In this case, the lower byte of each rectangle field is used to store the nearest character position, while the upper byte stores the pixel remainder to be added after the character size is multiplied in. Non-character-boundary objects may only be created in the "FREE" tree mode of the Resource Construction Set (also called "PANEL" in RCS 2.0). You should use them only in programs which will run in a single ST screen mode, because pixel coordinates are not portable between resolutions.

The first real secret of object rectangles is that each OBX and OBY is specified RELATIVE to the X and Y coordinate of its parent object within the tree. This is the first property we have seen that is actually "inherited" from level to level within the tree.

The second secret is more subtle: Every object's rectangle must be entirely contained within the rectangle of its parent. This principle goes by the names "bounding rectangles" or "visual hierarchy". We'll see in a moment how useful it is when detecting mouse/object collisions.

HOW GEM DOES IT.

Knowing these secrets, and the linkage structure of object trees, we can deduce how a number of the GEM operations must work. For instance, consider objcoffset, which returns the

actual screen X and Y of an object. We can see now that simply loading the OBX and OBY fields of the object does not suffice: they only give the offset relative to the parent object. So, objcoffset must BEGIN with these values, and then work its way back up to the ROOT of the tree, adding in the offsets found at each level.

This can be done by following the OBNEXT links from the chosen object. Whenever OBNEXT points to an object whose OBTAIL points right back to the same location, then the new node is another level, or "parent" in the tree, and objcoffset adds its OBX and OBY into the running totals. When OBNEXT becomes NIL, then the ROOT has been reached and the totals are the values to return. (By the way, remember that the OBX and OBY of the ROOT are undefined until formcenter has been called for the tree. They are shown as zeroes in the sample trees.)

We can also figure out objcdraw. It works its way DOWN the tree, drawing each object as it comes to it. It, too, must keep a running X and Y variable, adding in object offsets as it descends tree levels (using OBHEAD), and subtracting them again as it returns from each level. Since the larger objects are nearer the ROOT, we can now see why they are drawn first, with smaller objects drawn later or "on top of" them.

If you write an application which needs to move portions of a dialog or screen with respect to each other, you can take advantage of inheritance of screen position in objcdraw. Simply by changing the OBX and/or OBY of an object, you can move it and its entire sub-tree to a new location in the dialog. For instance, changing the coordinates of the parent box of a set of radio buttons will cause all of the buttons to move along with it.

Objcdraw also gives us an example of the uses of visual hierarchy. Recall that a clipping rectangle is specified when calling objcdraw. At each level of the tree we know that all objects below are contained in the screen rectangle of the current object. If the current rectangle falls completely outside the specified clipping rectangle, we know immediately that we need not draw the object, or any of its descendents! This ability to ignore an entire subtree is called "trivial rejection".

Now it's rather easy to figure out objcfind. It starts out by setting its "object found" variable to NIL. It begins a "walk" through the entire object tree, following OBHEAD and OBNEXT links, and keeping a current X and Y, just like objcdraw.

At each node visited, it simply checks to see if the "mouse" X,Y specified in the call are inside the current

object's rectangle. If they are, that object becomes the found object, and the tree walk continues with the object's offspring, and then siblings. Notice how this checking of offspring makes sure that a smaller object nested within, i.e., below, a larger object is found correctly.

If the mouse X,Y position is not within the object being checked, then by visual hierarchy it cannot be within any of its offspring, either. Trivial rejection wins again, and the entire sub-tree is skipped! Objcfind moves on to the OBNEXT of the rejected object.

THOUGHT EXPERIMENTS

Thinking about the objcfind algorithm reveals some information about its performance, and a few tricks we may use in improving the appearance of dialogs and other object trees.

First consider the problem of a dialog which contains many objects. If we lay them all out "side-by-side", then they will all be immediate offspring of the ROOT object. In this situation, the trivial rejection method will gain nothing. The time objcfind takes to complete will vary linearly with the total number of objects. This is called an "Order N" process.

Suppose that instead we broke up the dialog into two areas with invisible boxes, then broke up each of these areas in a like fashion, and so on until we got down to the size of the individual selectable objects. The number of bottom level objects in this scheme is a power of two equal to the depth of the tree. Trivial rejection is used to its fullest in this case. It is called an "Order Log N" process, and is much more efficient for large numbers of objects.

In practice, the speed of the ST will allow you to ignore this distinction for most dialogs and other trees. But if you get into a situation when speed is critical in searching a large tree, remember that nesting objects can improve performance dramatically.

If you have been following closely, you may have also noticed a hole in the visual hierarchy rule. It says that all of a node's children must lie within its rectangle, but it does NOT guarantee that the children's rectangles will be disjoint, that is, not overlap one another. This peculiarity is the basis of several useful tricks.

First, remember that objcfind always tries to scan the entire tree. That is, it doesn't quit when it finds the first object on the given coordinates. As mentioned above, this normally guarantees that nested objects will be found.

Consider, however, what happens when the mouse coordinates are on a point where two or more objects AT THE SAME LEVEL overlap: they will replace one another as the "found object" until objcfind returns with the one which is "last", that is, rightmost in the tree.

This quirk can be used to advantage in a number of cases. Suppose that you have in a dialog an image and a string which you would like to be selected together when either is clicked. Nesting within a common parent achieves nothing in this case. Instead, knowing that formdo must use objcfind, you could use our trick.

You have to know that the Resource Construction Set normally adds objects in a tree left to right, in the order in which you inserted them. You proceed to build the dialog in the following order: insert the image first, the string next, then carefully add an invisible box which is not nested within either, and size it to cover them both. Set the SELECTABLE attribute for the box, and the dialog manager will find it, and invert the whole area, when either the image or string is clicked.

By the way, remember that the SORT option in the RCS will change the order of an object's offspring. If you are going to try this trick, don't use SORT! It will undo all of your careful work.

A TREEWALKER OF OUR OWN

Since the GEM system gets so much mileage out of walking object trees, it seems reasonable that the same method should be useful in application programs. In the download you will find maptree(). As many LISP veterans might guess from the name, this code will traverse all or part of an object tree, applying a function to each node. It also allows the function to return a true/false value specifying whether the sub-tree below a particular node should be ignored. Let's examine maptree() in more detail as a final review of object tree structure.

First, look at the parameters. "tree" is the long address of the object tree of interest, as retrieved by rsrcgaddr. "this" is the node at which to begin the traverse, and "last" is the node at which to terminate.

In most cases, the beginning node will be ROOT, and the final value will be NIL. This will result in the entire tree being traversed. You may use other values, but be sure that you CAN get to "last" from "this" by following tree links! Although maptree() includes a safety check to prevent "running off" the tree, you could get some very strange results from incorrect

parameters.

The declaration for the final parameter, "routine", makes use of C construct which may be new to some. It is a pointer to a subroutine which returns a WORD as a result.

Maptree() begins by initializing a temporary variable, tmp1, which is used to store the number of the last node visited. Since no node will follow itself, setting tmp1 to the starting node is safe.

The main loop of the routine simply repeats visiting a new node until the last value is reached, or the safety check for end of tree is satisfied.

At any node visited, we can be in one of two conditions. Either we are at a node which is "new", that is, not previously visited, or else we are returning to a parent node which has already been processed. We can detect the latter condition by comparing the last node visited (tmp1) with the OBTAIL pointer of the current node. If the node is "old", it is not processed a second time, we simply update tmp1 and continue.

If the node is new, we call "routine" to process it, sending the tree address and object number as parameters. If a FALSE is returned, we will ignore any subtree below this node. On a TRUE return, we load up the OBHEAD pointer and follow it if a subtree exists. (If you don't care about rejecting subtrees, simply remove the if condition.) Finally, if the new node had no subtree, or was rejected by "routine", we follow along its OBNEXT link to the next node.

A simple application of our new tool shows its power. From a previous column you may recall the tedium of deselecting every button inside a dialog after it was completed. Using maptree(), you can deselect EVERY OBJECT in the tree by using maptree(tree, ROOT, NIL, deselobj); You must use a slightly modified version of deselobj() (included in the download) which always returns TRUE. Be sure to define or declare deselobj() in your code BEFORE making the maptree call!

In future columns, I will return to maptree() and show how it can be used for advanced techniques such as animated dialogs. In the meantime, experiment and enjoy!

PART - VI

Raster operations

SEASONS GREETINGS

This is the Yuletide installment of ST PRO GEM, devoted to explaining the raster, or "bit-blit" portion of the Atari ST's VDI functions.

Please note that this is NOT an attempt to show how to write directly to the video memory, although you will be able to deduce a great deal from the discussion.

As usual, there is a download with this column. You will find it in ATARI16 (PCS-58) in DL3 under the name of GEMCL6.C.

DEFINING TERMS

To understand VDI raster operations, you need to understand the jargon used to describe them. (Many programmers will be tempted to skip this section and go directly to the code. Please don't do it this time: Learning the jargon is the larger half of understanding the raster operations!)

In VDI terms a raster area is simply a chunk of contiguous words of memory, defining a bit image. This chunk is called a "form". A form may reside in the ST's video map area or it may be in the data area of your application. Forms are roughly analogous to "blits" or "sprites" on other systems. (Note, however, that there is no sprite hardware on the ST.)

Unlike other systems, there is NO predefined organization of the raster form. Instead, you determine the internal layout of the form with an auxiliary data structure called the MFDB, or Memory Form Definition Block. Before going into the details of the MFDB, we need to look at the various format options. Their distinguishing features are monochrome vs. color, standard vs. device-specific and even-word vs. fringed.

MONOCHROME VS. COLOR

Although these terms are standard, it might be better to say "single-color vs. multi-color". What we are actually defining is the number of bits which correspond to each dot, or pixel, on the screen. In the ST, there are three possible answers. The high-resolution mode has one bit per pixel, because there is only one "color": white.

In the medium resolution color mode, there are four possible colors for each pixel. Therefore, it takes two bits to represent each dot on the screen. (The actual colors which appear are determined by the settings of the ST's pallette registers.)

In the low resolution color mode, sixteen colors are generated requiring four bits per pixel. Notice that as the number of bits per pixel has been doubled for each mode, so the number of pixels on the screen has been halved: 640 by 400 for monochrome, 640 by 200 for medium-res, and 320 by 200 by low-res. In this way the ST always uses the same amount of video RAM: 32K.

Now we have determined how many bits are needed for each pixel, but not how they are laid out within the form. To find this out, we have to see whether the form is device-dependent or not.

STANDARD VS. DEVICE-SPECIFIC FORMAT

The standard raster form format is a constant layout which is the same for all GEM systems. A device-specific form is one which is stored in the internal format of a particular GEM system. Just as the ST has three different screen modes, so it has three different device-specific form formats. We will look at standard form first, then the ST-specific forms.

First, it's reasonable to ask why a standard format is used. Its main function is to establish a portability method between various GEM systems. For instance, an icon created in standard format on an IBM PC GEM setup can be moved to the ST, or a GEM Paint picture from an AT&T 6300 could be loaded into the ST version of Paint.

The standard format has some uses even if you only work with the ST, because it gives a method of moving your application's icons and images amongst the three different screen modes. To be sure, there are limits to this. Since there are different numbers of pixels in the different modes, an icon built in the high-resolution mode will appear twice as large in low-res mode, and would appear oblong in medium-res. (You can see this effect in the ST Desktop's icons.) Also, colors defined in the lower resolutions will be useless in monochrome.

The standard monochrome format uses a one-bit to represent black, and uses a zero for white. It is assumed that the form begins at the upper left of the raster area, and is written a word at a time left to right on each row, with the rows being

output top to bottom. Within each word, the most significant bit is the left-most on the screen.

The standard color form uses a storage method called "color planes". The high-order bits for all of the pixels are stored just as for monochrome, followed by the next-lowest bit in another contiguous block, and so on until all of the necessary color bits have been stored.

For example, on a 16-color system, there would be four different planes. The color of the upper-leftmost bit in the form would be determined by concatenating the high-order bit in the first word of each plane of the form.

The system dependent form for the ST's monochrome mode is very simple: it is identical to the standard form! This occurs because the ST uses a "reverse-video" setup in monochrome mode, with the background set to white.

The video organization of the ST's color modes is more complicated. It uses an "interleaved plane" system to store the bits which make up a pixel. In the low-resolution mode, every four words define the values of 16 pixels. The high-order bits of the four words are merged to form the left-most pixel, followed by the next lower bit of each word, and so on. This method is called interleaving because the usually separate color planes described above have been shuffled together in memory.

The organization of the ST's medium-resolution mode is similar to low-res, except the only two words are taken at a time. These are merged to create the two bits needed to address four colors.

You should note that the actual color produced by a particular pixel value is NOT fixed. The ST uses a color remapping system called a palette. The pixel value in memory is used to address a hardware register in the palette which contains the actual RGB levels to be sent to the display. Programs may set the palette registers with BIOS calls, or the user may alter its settings with the Control Panel desk accessory. Generally, palette zero (background) is left as white, and the highest numbered palette is black.

EVEN-WORD VS. FRINGES

A form always begins on a word boundary, and is always stored with an integral number of words per row. However, it is possible to use only a portion of the final word. This partial word is called a "fringe". If, for instance, you had a form 40 pixels wide, it would be stored with four words per row: three whole words, and one word with the eight pixel fringe

in its upper byte.

MFDB's

Now we can intelligently define the elements of the MFDB. Its exact C structure definition will be found in the download. The fdnplanes entry determines the color scheme: a value of one is monochrome, more than one denotes a color form. If fdstand is zero, then the form is device-specific, otherwise it is in standard format.

The fdw and fdh fields contain the pixel width and height of the form respectively. Fdwdwidth is the width of a row in words. If fdw is not exactly equal to sixteen times fdwdwidth, then the form has a fringe.

Finally, fdaddr is the 32-bit memory address of the form itself. Zero is a special value for fdaddr. It denotes that this MFDB is for the video memory itself. In this case, the VDI substitutes the actual address of the screen, and it ignores ALL of the other parameters. They are replaced with the size of the whole screen and number of planes in the current mode, and the form is (of course) in device-specific format.

This implies that any MFDB which points at the screen can only address the entire screen. This is not a problem, however, since the the VDI raster calls allow you to select a rectangular region within the form. (A note to advanced programmers: If this situation is annoying, you can retrieve the address of the ST's video area from low memory, add an appropriate offset, and substitute it into the MFDB yourself to address a portion of the screen.)

LET'S OPERATE

Now we can look at the VDI raster operations themselves. There are actually three: transform form, copy raster opaque, and copy raster transparent. Both copy raster functions can perform a variety of logic operatoins during the copy.

TRANSFORM FORM

The purpose of this operation is to change the format of a form: from standard to device-specific, or vice-versa. The calling sequence is:

```
vrtrnfm(vdihandle, source, dest);
```

where source and dest are each pointers to MFDBs. They ARE

allowed to be the same. Transform form checks the fdstand flag in the source MFDB, toggles it and writes it into the destination MFDB after rewriting the form itself. Note that transform form CANNOT change the number of color planes in a form: fdnplanes must be identical in the two MFDBs.

If you are writing an application to run on the ST only, you will probably be able to avoid transform form entirely. Images and icons are stored within resources as standard forms, but since they are monochrome, they will work "as is" with the ST.

If you may want to move your program or picture files to another GEM system, then you will need transform form. Screen images can be transformed to standard format and stored to disk. Another system with the same number of color planes could the read the files, and transform the image to ITS internal format with transform form.

A GEM application which will be moved to other systems needs to contain code to transform the images and icons within its resource, since standard and device-specific formats will not always coincide.

If you are in this situation, you will find several utilities in the download which you can use to transform GICON and GIMAGE objects. There is also a routine which may be used with maptree() from the last column in order to transform all of the images and icons in a resource tree at once.

COPY RASTER OPAQUE

This operation copies all or part of the source form into the destination form. Both the source and destination forms must be in device-specific form. Copy raster opaque is for moving information between "like" forms, that is, it can copy from monochrome to monochrome, or between color forms with the same number of planes. The calling format is:

```
vrocpyfm(vdihandle, mode, pxy, source, dest);
```

As above, the source and dest parameters are pointers to MFDBs (which in turn point to the actual forms). The two MFDBs may point to memory areas which overlap. In this case, the VDI will perform the move in a non-destructive order. Mode determines how the pixel values in the source and destination areas will be combined. I will discuss it separately later on.

The pxy parameter is a pointer to an eight-word integer array. This array defines the area within each form which will be affected. Pxy[0] and pxy[1] contain, respectively, the X and

Y coordinates of the upper left corner of the source rectangle. These are given as positive pixel displacements from the upper left of the form. Pxy[2] and pxy[3] contain the X and Y displacements for the lower right of the source rectangle.

Pxy[4] through pxy[7] contain the destination rectangle in the same format. Normally, the destination and source should be the same size. If not, the size given for the source rules, and the whole are is transferred beginning at the upper left given for the destination.

This all sounds complex, but is quite simple in many cases. Consider an example where you want to move a 32 by 32 pixel area from one part of the display to another. You would need to allocate only one MFDB, with a zero in the fdaddr field. The VDI will take care of counting color planes and so on. The upper left raster coordinates of the source and destination rectangles go into pxy[0], pxy[1] and pxy[4], pxy[5] respectively. You add 32 to each of these values and insert the results in the corresponding lower right entries, then make the copy call using the same MFDB for both source and destination. The VDI takes care of any overlaps.

COPY RASTER TRANSPARENT

This operation is used for copying from a monochrome form to a color form. It is called transparent because it "writes through" to all of the color planes. Again, the forms need to be in device-specific form. The calling format is:

```
vrtcpyfm(vdihandle, mode, pxy, source, dest, color);
```

All of the parameters are the same as copy opaque, except that color has been added. Color is a pointer to a two word integer array. Color[0] contains the color index which will be used when a one appears in the source form, and color[1] contains the index for use when a zero occurs.

Incidentally, copy transparent is used by the AES to draw GICONs and GIMAGEs onto the screen. This explains why you do not need to convert them to color forms yourself.

A note for advanced VDI programmers: The pxy parameter in both copy opaque and transparent may be given in normalized device coordinates (NDC) if the workstation associated with vdihandle was opened for NDC work.

THE MODE PARAMETER

The mode variable used in both of the copy functions is an

integer with a value between zero and fifteen. It is used to select how the copy function will merge the pixel values of the source and destination forms. The complete table of functions is given in the download. Since a number of these are of obscure or questionable usefulness, I will only discuss the most commonly used modes.

REPLACE MODE

A mode of 3 results in a straight-forward copy: every destination pixel is replaced with the corresponding source form value.

ERASE MODE

A mode value of 4 will erase every destination pixel which corresponds to a one in the source form. (This mode corresponds to the "eraser" in a Paint program.) A mode value of 1 will erase every destination pixel which DOES NOT correspond to a one in the source.

XOR MODE

A mode value of 6 will cause the destination pixel to be toggled if the corresponding source bit is a one. This operation is invertable, that is, executing it again will reverse the effects. For this reason it is often used for "software sprites" which must be shown and then removed from the screens. There are some problems with this in color operations, though - see below.

TRANSPARENT MODE

Don't confuse this term with the copy transparent function itself. In this case it simply means that ONLY those destination pixels corresponding with ones in the source form will be modified by the operation. If a copy transparent is being performed, the value of color[0] is substituted for each one bit in the source form. A mode value of 7 selects transparent mode.

REVERSE TRANSPARENT MODE

This is like transparent mode except that only those destination pixels corresponding to source ZEROS are modified. In a copy transparent, the value of color[1] is substituted for each zero bit. Mode 13 selects reverse transparent.

THE PROBLEM OF COLOR

I have discussed the various modes as if they deal with one and zero pixel values only. This is exactly true when both forms are monochrome, but is more complex when one or both are color forms.

When both forms are color, indicating that a copy opaque is being performed, then the color planes are combined bit-by-bit using the rule for that mode. That is, for each corresponding source and destination pixel, the VDI extracts the top order bits and processes them, then operates on the next lower bit, and so on, stuffing each bit back into the destination form as the copy progresses. For example, an XOR operation on pixels valued 7 and 10 would result in a pixel value of 13.

In the case of a copy transparent, the situation is more complex. The source form consists of one plane, and the destination form has two or more. In order to match these up, the color[] array is used. Whenever a one pixel is found, the value of color[0] is extracted and used in the bit-by-bit merge process described in the last paragraph. When a zero is found, the value of color[1] is merged into the destination form.

As you can probably see, a raster copy using a mode which combines the source and destination can be quite complex when color planes are used! The situation is compounded on the ST, since the actual color values may be remapped by the palette at any time. In many cases, just using black and white in color[] may achieve the effects you desire. If need to use full color, experimentation is the best guide to what looks good on the screen and what is garish or illegible.

OPTIMIZING RASTER OPERATIONS

Because the VDI raster functions are extremely generalized, they are also slower than hand-coded screen drivers which you might write for your own special cases. If you want to speed up your application's raster operations without writing assembl language drivers, the following hints will help you increase the VDI's performance.

AVOID MERGED COPIES

These are copy modes, such as XOR, which require that words be read from the destination form. This extra memory access increases the running time by up to fifty percent.

MOVE TO CORRESPONDING PIXELS

The bit position within a word of the destination rectangle should correspond with the bit position of the source rectangle's left edge. For instance, if the source's left edge is one pixel in, then the destination's edge could be at one, seventeen, thirty-three, and so. Copies which do not obey this rule force the VDI to shift each word of the form as it is moved.

AVOID FRINGES

Put the left edge of the source and destination rectangles on an even word boundary, and make their widths even multiples of sixteen. The VDI then does not have to load and modify partial words within the destination forms.

USE ANOTHER METHOD

Sometimes a raster operation is not the fastest way to accomplish your task. For instance, filling a rectangle with zeros or ones may be accomplished by using raster copy modes zero and fifteen, but it is faster to use the VDI vbar function instead. Likewise, inverting an area on the screen may be done more quickly with vbar by using BLACK in XOR mode. Unfortunately, vbar cannot affect memory which is not in the video map, so these alternatives do not always work.

FEEDBACK RESULTS

The results of the poll on keeping or dropping the use of portability macros are in. By a slim margin, you have voted to keep them. The vote was close enough that in future columns I will try to include ST-only versions of routines which make heavy use of the macros. C purists and dedicated Atarians may then use the alternate code.

THE NEXT QUESTION

This time I'd like to ask you to drop by the Feedback Section and tell me whether the technical level of the columns has been:

A) Too hard! Who do you think we are, anyway?
B) Too easy! Don't underestimate Atarians.
C) About right, on the average.

If you have the time, it would also help to know a little about your background, for instance, whether you are a professional programmer, how long you have been computing, if you owned an 8-bit Atari, and so on.

COMING UP SOON

The next column will deal with GEM menus: How they are constructed, how to decipher menu messages, and how to change menu entries at run-time. The following issue will contain more feedback response, and a discussion on designing user interfaces for GEM programs.

PART - VII

Menu Structures

HAPPY NEW YEAR

This is article number seven in the ST PRO GEM series, and the first for 1986. In this installment, I will be discussing GEM menu structures and how to use them in your application. There is also a short Feedback response section. You will find the download file containing the code for this column in the file GEMCL7.C in DL3 of the ATARI16 SIG (PCS-58).

MENU BASICS

In ST GEM, the menu consists of a bar across the top of the screen which displays several sub-menu titles. Touching one of the titles causes it to highlight, and an associated "drop-down" to be drawn directly below on the screen. This drop-down may be dismissed by moving to another title, or by clicking the mouse off of the drop-down.

To make a selection, the mouse is moved over the drop-down. Each valid selection is highlighted when the mouse touches it. Clicking the mouse while over one of these selections picks that item. GEM then undraws the drop-down, and sends a message to your application giving the object number of the title bar entry, and the object number of the drop-down item which were selected by the user. The selected title entry is left highlighted while your code processes the request.

MENU STRUCTURES

The data structure which defines a GEM menu is (surprise!) an object tree, just like the dialogs and panels which we have discussed before. However, the operations of the GEM menu manager are quite different from those of the form manager, so the internal design of the menu tree has some curious constraints.

The best way to understand these constraints is to look at an example. The first item in the download is the object structure (only) of the menu tree from the GEM Doodle/Demo sample application.

The ROOT of a menu tree is sized to fit the entire screen. To satisfy the visual hierarchy principle (see article #5), the screen is divided into two parts: THE BAR, containing the menu

titles, and THE SCREEN, while contains the drop-downs when they are drawn. Each of these areas is defined by an object of the same name, which are the only two objects linked directly below the ROOT of a menu tree. You will notice an important implication of this structure: The menu titles and their associated drop-downs are stored in entirely different subtrees of the menu!

While examining THE BAR in the example listing, you may notice that its OBHEIGHT is very large (513). In hexadecimal this is 0x0201. This defines a height for THE BAR of one character plus two pixels used for spacing. THE BAR and its subtree are the only objects which are drawn on the screen in the menu's quiescent state.

The only offspring object of THE BAR is THE ACTIVE. This object defines the part of THE BAR which is covered by menu titles. The screen rectangle belonging to THE ACTIVE is used by the GEM screen manager when it waits for the mouse to enter an active menu title. Notice that THE ACTIVE and its offspring also have OBHEIGHTs with pixel residues.

The actual menu titles are linked left to right in order below THE ACTIVE. Their OBXs and OBWIDTHs are arranged so that they completely cover THE ACTIVE. Normally, the title objects are typed GTITLE, a special type which assures that the title bar margins are correctly drawn.

THE SCREEN is the parent object of the drop-down boxes themselves. They are linked left to right in an order identical with their titles, so that the menu manager can make the correct correspondence at run-time. The OBX of each drop-down is set so that it is positioned below its title on the screen.

Notice that it is safe to overlap the drop-downs within a menu, since only one of them will be displayed at any time. There is one constraint on the boxes however: They must be no greater than a quarter screen in total size. This is the size of the off-screen blit buffer which is used by GEM to store the screen contents when the drop-down is drawn. If you exceed this size, not all the screen under the drop-down will be restored, or the ST may crash!

The entries within a drop-down are usually GSTRINGs, which are optimized for drawing speed. The rectangles of these entries must completely cover the drop-down, or the entire drop-down will be inverted when the mouse touches an uncovered area! Techniques for using objects other than GSTRINGs are discussed later in this column.

The first title and its corresponding drop-down are special. The title name, by custom, is set to DESK. The

drop-down must contain exactly eight GSTRING objects. The first (again by custom) is the INFO entry, which usually leads to a dialog displaying author and copyright information for your application. The next is a separator string of dashes with the DISABLED flag set. The following six objects are dummy strings which GEM fills in with the names of desk accessories when your menu is loaded.

The purpose of this description of menu trees is to give you an understanding of what lies "behind the scenes" in the next section, which describes the run-time menu library calls. In practice, the Resource Construction Set provides "blank menus" which include all of the required elements, and it also enforces the constraints on internal structure. You only need to worry about these if you modify the menu tree "on-the-fly".

USING THE MENU

once you have loaded the application's resource, you can ask the AES to install your menu. You must first get the address of the menu tree within the resource using:

```
rsrcgaddr(RTREE, MENUTREE, &admenu);
```

assuming that MENUTREE is the name you gave the menu in the RCS, and that admenu is a LONG which will receive the address. Then you call the AES to establish the menu:

```
menubar(admenu, TRUE);
```

At this point, the AES draws your menu bar on the screen and animates it when the user moves the mouse into the title area.

The AES indicates that the user has made a menu selection by sending your application a message. The message type is MNSELECTED, which will be stored in msg[0], the first location in the message returned by evntmulti().

The AES also stores the object number of the selected menu's title in msg[3], and the object number of the selected menu item in msg[4]. Generally, your application will process menu messages with nested C switch statements. The outer switch will have one case for each menu title, and the inner switch statements will have a case for each entry within the selected menu. (This implies that you must give a name to each title and to each menu entry when you create the menu in the RCS.)

After the user has made a menu selection, the AES leaves the title of the chosen menu in reverse video to indicate that your application is busy processing the message. When you done with whatever action is indicated, you need to return the title

to a normal state. This is done with

```
menutnormal(admenu, msg[3], TRUE);
```

(Remember that msg[3] is the title's object number.)

When your application is ready to terminate, it should delete its menu bar. Do this with the call: menubar(admenu, FALSE);

GETTING FANCY

The techniques above represent the bare minimum to handle menus. In most cases, however, you will want your menus to be more "intelligent" in displaying the user's options. For instance, you can prevent many user errors by disabling inappropriate choices, or you can save space on drop-downs by showing only one line for a toggle and altering its text or placing and removing a check mark when the state is changed. This section discusses these and other advanced techniques.

It is a truism of user interface design that the best way to deal with an error is not to let it happen in the first place. It many cases, you can apply this principle to GEM menus by disabling choices which should not be used. If your application uses a "selection precedes action" type of interface, the type of object selected may give the information needed to do this. Alternately, the state of the underlying program may render certain menu choices illegal.

GEM provides a call to disable and re-enable menu options. The call is:

```
menuienable(admenu, ENTRY, FALSE);
```

to disable a selection. The entry will be grayed out when it is drawn, and will not invert under the mouse and will not be selected by the user. Substituting TRUE for FALSE re-enables the option. ENTRY is the name of the object which is being affected, as assigned in the RCS.

Note that menuienable() will not normally affect the appearance or operation of menu TITLE entries. However, there is an undocumented feature which allows this. If ENTRY is replaced by the object number of a title bar entry with its top bit set, then the entire associated drop-down will be disabled or re-enabled as requested, and the title's appearance will be changed. But, be warned that this feature did not work reliably in some early versions of GEM. Test it on your copy of ST GEM, and use it with caution when you cannot control the version under which your application may run.

It is also possible to disable menu entries by directly altering the DISABLED attribute within the OBSTATE word. The routines enabobj() and disabobj() in the download show how this is done. They are also used in setmenu(), which follows them immediately.

Setmenu() is a utility which is useful when you wish to simultaneously enable or disable many entries in the menu when the program's state changes or a new object is selected by the user. It is called with

```
setmenu(admenu, vector);
```

where vector is a pointer to an array of WORDs. The first word of the array determines the default state of menu entries. If it is TRUE, then setmenu() enables all entries in every drop-down of the menu tree, except that the DESK drop-down is unaffected. If it is FALSE, then every menu entry is disabled.

The following entries in the array are the numbers of menu entries which are to be toggled to the reverse of the default state. This list is terminated by a zero entry.

The advantage of setmenu() is that it allows you to build a collection of menu state arrays, and associate one with each type of user-selected object, program state, and so on. Changing the status of the menu tree may then be accomplished with a single call.

CHECK PLEASE?

One type of state indicator which may appear within a drop-down is a checkmark next to an entry. You can add the checkmark with the call:

```
menuicheck(admenu, ENTRY, TRUE);
```

and remove it by replacing the TRUE with FALSE. As above, ENTRY is the name of the menu entry of interest. The checkmark appears inside the left boundary of the entry object, so leave some space for it.

The menuicheck() call is actually changing the state of the CHECKED flag within the entry object's OBSTATE word. If necessary, you may alter the flag directly using doobj() and undoobj() from the download.

NOW YOU SEE IT NOW YOU DON'T

You can also alter the text which appears in a particular menu entry (assuming that the entry is a GSTRING object). The call

```
menutext(admenu, ENTRY, ADDR(text));
```

will substitute the null-terminated string pointed to by text for whatever is currently in ENTRY. Remember to make the drop-down wide enough to handle the largest text string which you may substitute. In the interests of speed, GSTRINGs drawn within drop-downs are not clipped, so you may get garbage characters on the desktop if you do not size the drop-down properly!

The menutext() call actually alters the OBSPEC field of the menu entry object to point to the string which you specify. Since the menu tree is a static data structure which may be directly accessed by the AES at any time, be sure that the string is also statically allocated and that it is not modified without first being delinked from the menu tree. Failure to do this may result in random crashes when the user accesses the drop-down!

LUNCH AND DINNER MENUS

Some applications may have such a wide range of operations that they need more than one menu bar at different times. There is no problem with having more than one menu tree in a resource, but the AES can only keep track of one at a time. Therefore, to switch menus you need to use menubar(admenu1, FALSE); to release the first menu, then use menubar(admenu2, TRUE); to load the second menu tree.

Changing the entire menu is a drastic action. Out of consideration for your user, it should be associated with some equally obvious change in the application which has just been manually requested. An example might be changing from spreadsheet to data graphing mode in a multi-function program.

DO IT YOURSELF

In a future column, I will discuss how to set up user-defined drawing objects. If you have already discovered them on your own, you can use them within a drop-down or as a title entry.

If the user-defined object is within a drop-down, its associated drawing code will be called once when the drop-down is first drawn. It will then be called in "state-change" mode when the entry is highlighted (inverted). This allows you to use non-standard methods to show selection, such as outlines.

If you try to insert a user-defined object within the menu title area, remember that the GTITLE object which you are replacing includes part of the dark margin of the bar. You will need to experiment with your object drawing code to replicate this effect.

MAKE PRETTY

There are a number of menu formatting conventions which have become standard practice. Using these gives your application a recognizable "look-and-feel" and helps users learn it. The following section reviews these conventions, and supplies a few hints and tricks to obtain a better appearance for you menus.

The second drop-down is customarily used as the FILE menu. It contains options related to loading and saving the files used by the application, as well as entries for clearing the workspace and terminating the program.

You should avoid crowding the menu bar. Leave a couple of spaces between each entry, and try not to use more than 70% of the bar. Not only does this look better, but you will have space for longer words if you translate your application to a foreign language.

Similarly, avoid cluttering menu drop-downs. Try to keep the number of options to no more than ten unless they are clearly related, such as colors. Separate off dissimilar entries with the standard disabled dashes line. (If you are using setmenu(), remember to consider the separators when setting up the state vectors.)

If the number of options grows beyond this bound, it may be time to move them to a dialog box. If so, it is a convention to put three dots following each menu entry which leads to a dialog. Also, allow a margin on the menu entries. Two leading blanks and a minimum of one trailing blank is standard, and allows room for checkmarks if they are used.

Dangerous menu options should be far away from common used entries, and are best separated with dashed lines. Such options should either lead to a confirming go/no-go alert, or should have associated "undo" options.

After you have finished defining a menu drop-down with the RCS, be sure that its entries cover the entire box. Then use ctrl-click to select the drop-down itself, and SORT the entries top to bottom. This way the drop-down draws in smoothly top to bottom.

Finally, it is possible to put entries other than GSTRINGs into drop-downs. In the RCS, you will need to import them via the clipboard from the Dialog mode.

Some non-string object, such as icons and images, will look odd when they are inverted under the mouse. There is a standard trick for dealing with this problem. Insert the icon or whatever in the drop-down first. Then get a GIBOX object and position and size it so that it covers the first object as well as the extra area you would like to be inverted.

Edit the GIBOX to remove its border, and assign the entry name to it. Since the menu manager uses objcfind(), it will detect and invert this second object when the mouse moves into the drop-down. (To see why, refer to article #5.) Finally, DO NOT SORT a drop-down which has been set up this way!

THAT'S IT FOR NOW!

The next column will discuss some of the principles of designing GEM interfaces for applications. This topic is irreverantly known as GEM mythology or interface religion. The subject for the following column is undecided. I am considering mouse and keyboard messages, VDI drawing primitives, and the file selector as topics. Let me know your preferences in the Feedback!

PART - VIII

USER INTERFACES

AND NOW FOR SOMETHING COMPLETELY DIFFERENT!

In response to a number of requests, this installment of ST PRO GEM will be devoted to examining a few of the principles of computer/human interface design, or "religion" as some would have it. I'm going to start with basic ergonomic laws, and try to draw some conclusions which are fairly specific to designing for the ST. If this article meets with general approval, further "homilies" may appear at irregular intervals as part of the ST PRO GEM series.

For those who did NOT ask for this topic, it seems fair to explain why your diet of hard-core technical information has been interrupted by a sermon! As a motivater, we might consider why some programs are said by reviewers to have a "hot" feel (and hence sell well!) while others are "confusing" or "boring".

Alan Kay has said that "user interface is theatre". I think we may be able to take it further, and suggest that a successful program works a bit of magic, persuading the user to suspend his disbelief and enter an imaginary world behind the screen, whether it is the mathematical world of a spreadsheet, or the land of Pacman pursued by ghosts.

A reader of a novel or science fiction story also suspends disbelief to participate in the work. Bad grammar and clumsy plotting by the author are jarring, and break down the illusion. Similarly, a programmer who fails to pay attention to making his interface fast and consistent will annoy the user, and distract him from whatever care has been lavished on the functional core of the program.

CREDIT WHERE IT'S DUE

Before launching into the discussion of user interface, I should mention that the general treatment and many of the specific research results are drawn from Card, Newell, and Moran's landmark book on the topic, which is cited at the end of the article. Any errors in interpretation and application to GEM and the ST are entirely my own, however.

FINGERTIPS

We'll start right at the user's fingers with the basic equation governing positioning of the mouse, Fitt's Law, which is given as

$$T = I * LOG2(D / S + .5)$$

where T is the amount of time to move to a target, D is the distance of the target from the current position, and S is the size of the target, stated in equivalent units. LOG2 is the base 2 (binary) logarithm function, and I is a proportionality constant, about 100 milliseconds per bit, which corresponds to the human's "clock rate" for making incremental movements.

We can squeeze an amazing amount of information out of this formula when attempting to speed up an interface. Since motion time goes up with distance, we should arrange the screen with the usual working area near the center, so the mouse will have to move a smaller distance on average from a selected object to a menu or panel. Likewise, any items which are usually used together should be placed together.

The most common operations will have the greater impact on speed, so they should be closest to the working area and perhaps larger than other icons or menu entries. If you want to have all other operations take about the same time, then the targets farthest from the working area should be larger, and those closer may be proportionately smaller.

Consider also the implications for dialogs. Small check boxes are out. Large buttons which are easy to hit are in. There should be ample space between selectable items to allow for positioning error. Dangerous options should be widely separated from common selections.

MUSCLES

Anyone who has used the ST Desktop for any period of time has probably noticed that his fingers now know where to find the File menu. This phenomenon is sometimes called "muscle memory", and its rate of onset is given by the Power Law of Practice:

$$T(n) = T(1) * n ** (-a)$$

where T(n) is the time on the nth trial, T(1) is the time on the first trial, and a is approximately 0.4. (I have appropriated ** from Fortran as an exponentiation operator, since C lacks one.)

This first thing to note about the Power Law is that it only works if a target stays in the same place! This should be a potent argument against rearranging icons, menus, or dialogs without some explicit request by the user. The time to hit a target which moves around arbitrarily will always be T(1)!

In many cases, the Power Law will also work for sequences of operations to even greater effect. If you are a touch typist, you can observe this effect by comparing how fast you can enter "the" in comparison to three random letters. We'll come back shortly to consider what we can do to encourage this phenomenon.

EYES

Just as fingers are the way the user sends data to the computer, so the eyes are his channel from the machine. The rate at which information may be passed to the user is determined by the "cycle time" of his visual processor. Experimental results show that this time ranges between 50 and 200 milliseconds.

Events separated by 50 milliseconds or less are always perceived as a single event. Those separated by more than 200 milliseconds are always seen as separate. We can use these facts in optimizing user of the computer's power when driving the interface.

Suppose your application's interface contains an icon which should be inverted when the mouse passes over it. We now know that flipping it within one twentieth of a second is necessary and sufficient. Therefore, if a "first cut" at the program achieves this performance, there is no need for further optimization, unless you want to interleave other operations. If it falls short, it will be necessary to do some assembly coding to achieve a smooth feel.

On the other hand, two actions which you want to appear distinct or convey two different pieces of information must be separated by an absolute minimum of a fifth of a second, even assuming that they occur in an identical location on which the user's attention is already focused.

We are able to influence the visual processing rate within the 50 to 200 millisecond range by changing the intensity of the stimulus presented. This can be done with color, by flashing a target, or by more subtle enhancements such as bold face type. For instance, most people using GEM soon become accustomed to the "paper white" background of most windows and dialogs. A dialog which uses a reverse color scheme, white letters on

black, is visually shocking in its starkness, and will immediately draw the user's eyes.

It should be quickly added that stimulus enhancement will only work when it unambiguously draws attention to the target. Three or four blinking objects scattered around the screen are confusing, and worse than no enhancement at all!

SHORT-TERM MEMORY

Both the information gathered by the eyes and movement commands on their way to the hand pass through short-term memory (also called working memory). The amount of information which can be held in short-term memory at any one time is limited. You can demonstrate this limit on yourself by attempting to type a sheet of random numbers by looking back and forth from the numbers to the screen. If you are like most people, you will be able to remember between five and nine numbers at a time. So universal is this finding that it is sometimes called "the magic number seven, plus or minus two".

This short-term capacity sets a limit on the number of choices which the user can be expected to grasp at once. It suggests that the number of independent choices in a menu, for instance, should be around seven, and never exceed nine. If this limit is violated, then the user will have to take several glances, with pauses to think, in order to make a choice.

CHUNKING

The effective capacity of short-term memory can be increased when several related items are mentally grouped as a "chunk". Humans automatically adopt this strategy to save themselves time. For instance, random numbers had to be used instead of text in the example above, because people do not type their native language as individual characters. Instead, they combine the letters into words and remember these chunks instead. Put another way, the characters are no longer considered as individual choices.

A well designed interface should promote the use of chunking as a strategy by the user. One easy way is to gather together related options in a single place. This is one reason that like commands are grouped into a single menu which is hidden except for its title. If all of the menu options were "in the open", the user would be overwhelmed with dozens of alternatives at once. Instead, a "Show Info" command, for instance, becomes two chunks: pick File menu, then pick Show.

Sometimes the interface can accomplish the chunking for the user. Consider the difference between a slider bar in a GEM program, and a three digit entry field in a text mode application. Obviously, the GEM user has fewer decisions to make in order to set the associated variable.

THINK!

While we are puttering around trying to speed up the keyboard, the mouse, and the screen, the user is actually trying to get some work done. We need to back off now, and look at the ways of thinking, or cognitive processes, that go into accomplishing the job.

The user's goal may be to enter and edit a letter, to retrieve information from a database, or simply draw a picture, but it probably has very little to do with programming. In fact, the Problem Space Principle says that the task can be described as a set of states of knowledge, a set of operators and associated constraints for changing the states, and the knowledge to choose the appropriate operator, which resides in the user's head.

Those with a background in systems theory can consider this as a somewhat abstract, but straightforward, statement in terms of state variables and operators. A programmer might compare the knowledge states to the values of variables, the operators to arithmetic and logic operations, the constraints to the rules of syntax, and the user's knowledge to the algorithm embodied by a program.

ARE WE NOT MEN?

A rational person will try to attain his goals (get the job done) by changing the state of his problem space from its initial state to the goal state. The initial state, for instance, might be a blank word processor screen. The desired final state is to have a completed business letter on the screen.

The Rationality Principle says that the user's behavior in typing, mousing, and so on, can be explained by considering the tasks required to achieve the goal, the operators available to carry out the tasks, and the limitations on the user's knowledge, observations, and processing capacity. This sounds like the typical user of a computer program must spend a good deal of time scratching his head and wondering what to do next. In fact, one of Card and Moran's key results is that this is NOT what takes place.

What happens, in fact, is that the trained user strikes a sort of "modus vivendi" with his tool and adopts a set of repetitive, trained behavior patterns as the best way to get the job done. He may go so far as to ignore some functions of the program in order to set up a reliable pattern. What we are looking for is a way of measuring and predicting the "quality" of this trained behavior. Since using computers is a human endeavor, we should consider not only the speed with which the task is completed, but the degree of annoyance or pleasure associated with the process.

Card and Moran constructed a series of behavioral models which they called GOMS models, for Goals-Operators-Methods-Selection. These models suggested that in the training process the user learned to combine the basic operators in sequences (chunks!) which then became methods for reaching the goals. Then these first level methods might be combined again into second level methods, and so forth, as the learning progressed.

The GOMS models were tested in a lengthy series of trials at Xerox PARC using a variety of word processing software. (Among the subjects of these experiments were the inventors of the windowing methods used in GEM!) The results were again surprising: the level of detail in the models was really unimportant!

It turned out to be sufficient to merely count up the number of keystrokes, mouse movements, and thought intervals required by each task. After summing up all of the tasks, any extra time for the computer to respond, or the user to move his hands from keyboard to mouse, or eyes from screen to printed page is added in. This simplified version is called the Keystroke-Level Model.

As an example of the Keystroke Model, consider the task of changing a mistyped letter on the screen of a GEM word processor. This might be broken down as follows: 1) find the letter on the screen; 2) move hand to mouse; 3) point to letter; 4) click mouse button; 5) move hand to keyboard; 6) strike "Delete" key; 7) strike key for new character.

The sufficiency of the Keystroke Model is great news for our attempt to design faster interfaces. It says we can concentrate our efforts on minimizing the number of total actions to be taken, and making sure that each action is as fast as possible. We have already discussed some ways to speed up the mouse and keyboard actions, so let's now consider how to speed up the thought intervals, and cut the number of actions.

One way to cut down "think time" is to make sure that the capacity of short-term memory is not exceeded during the course of a task. For example, the fix-a-letter task described above required the user to remember 1) his place in the overall job of typing the document; 2) the task he is about to perform; 3) where the bad character appeared, and 4) what the new character was. When this total of items creeps toward seven, the user often loses his place and commits errors.

You can appreciate the ubiquity of this problem by considering how many times you have made mistakes nesting parentheses, or had to go back to count them, because too many things happened while typing the line to remember the nesting levels. The moral is that operations with long strings of operands should be avoided when designing an interface.

The single most important factor in making an interface comfortable to use is increasing its predictability, and decreasing the amount of indecision present at each step during a task. There is (inevitably) an Uncertainty Principle which relates the number of choices at each step to the associated time for thought:

$$T = I * LOG2 (N + 1)$$

where LOG2 is the binary logarithm function, N is the number of equally probable choices, and I is a constant of approximately 140 msec/bit. When the alternates are not equally probable, the function is more complex:

$$T = I * SUM\text{-}FOR\text{-}i\text{-}FROM\text{-}1\text{-}TO\text{-}N\ (P(i) * LOG2(1 / P(i) + 1))$$

where the P(i) are the probabilities of each of the choices (which must sum to one). (SUM-FOR-i... is the best I can do for a sigma operator on-line!) Those of you with some information theory background will recognize this formula as the entropy of the decision; we'll come back to that later.

So what can we learn from this hash? It turns out, as we might expect, that we can decrease the decision time by making some of the user's choices more probable than others. We do that by means of feedback cues from the interface.

The important of reliable, continuous meaningful feedback cannot be emphasized enough. It helps the beginner learn the system, and its predictability makes the program comfortable for the expert. Programs with no feedback, or unreliable cues, produce confusion, dissonance, and frustration in the user.

This principle is so important that I going to give several examples from common GEM practice. The Desktop provides several

instances. When an object is selected and a menu drops down, only those choices which are legal for the object are in black. The others are dimmed to grey, and are therefore removed from the decision. When a pick is made from the menu, the bar entry remains black until the operation is complete, reassuring the user that the correct choice was made. In both the Desktop and the RCS, items which are double-clicked open up with a "zoom box" from the object, again showing that the right object was picked.

Other techniques are useful when operator icons are exposed on the screen. When an object is picked, the legal operations might be outlined, or the bad choices might be dimmed. If the screen flashing produced by this is objectionable, the legal icons can be made mouse sensitive, so they will "light up" when the cursor passes over - again showing the user which choices are legal.

The desire for feedback is so strong that it should be provided even while the computer is doing an operation on its own. The hour glass mouse form is a primitive example of this. More sophisticated are "progress indicators" such as animated thermometer bars, clocks, or text displays of the processing steps. The ST Desktop provides examples in the Format and Disk Copy functions. The purpose of all of these is to reassure the user that the operation is progressing normally. Their lack can lead to amusing spectacles such as secretaries leaning over to hear if their disk drives are working!

Another commonly overlooked feature is error prevention and correction. Card and Moran's results showed that in order to go faster, people will tolerate error rates of up to 30% in their work. Any program which does not give a fast way to fix mistakes will be frustrating indeed!

The best way to cope with an error is to "make it didn't happen", to quote a common child's phrase. The same feedback methods discussed above are also effective in preventing the user from picking inappropriate combinations of objects and operations. Replacement of numeric type-ins with sliders or other visual controls eliminates the common "Range Error". The use of radio buttons prevents the user from picking incompatible options. When such techniques are used consistently, the beginner also gains confidence that he may explore the program without blundering into errors.

Once an error has occured, the best solution is to have an "inverse operation" immediately available. For instance, the way to fix a bad character is to hit the backspace key. If a line is inadvertantly deleted, there should be a way to restore it.

Sometimes the mechanics of providing true inverses are impractical, or end up cluttering the interface themselves. In these cases, a global "Undo" command should be provided to reverse the effect of the last operation, no matter what it was.

OF MODES AND BANDWIDTH

Now I am going to depart from the Card, Newell and Moran thread of discussion to consider how we can minimize the number of operations in a task by altering the modes of the interface. Although "no modes" has been a watchword of Macintosh developers, the term may need definition for Atarians.

Simply stated, a mode exists any time you cannot get to all of the capabilities of the program without taking some intermediate step. Familiar examples are old-style "menu-driven" programs, in which user must make selections from a number of nested menus in order to perform any operation. The options of any one menu are unavailable from the others.

Recall that the user is trying to accomplish work in his own problem space, by altering its states. A mode in the program adds additional states to the problem space, which he is forced to consider in order to get the job done. We might call an interface which is completely modeless "transparent", because it adds no states between the user and his work. One of the best examples of a transparent program is the 15-puzzle in the Macintosh desk accessory set. The problem space of rearranging the tiles is identical between the program and a physical puzzle.

Unfortunately, most programmers find themselves forced to put modes of some sort into their programs. These often arise due to technological limitations, such as memory space, screen "real estate", or performance limitations of peripherals. The question is how the modes can be made least offensive.

I will make the general claim that the frustration which a mode produces is directly proportional to the amount of the user's bandwidth which it consumes. In other words, we need to consider how many keystrokes, mouse clicks, eye movements, and so on, are going into manipulating the true problem states, and how many are being absorbed by the modes of the program. If the interface is wasting a large amount of the user's effort, it will be perceived as slow and annoying.

Here we can consider again the hierarchy of goals and methods which the user employs. When the mode is low in the hierarchy, and close to the user's "fingertips", it is

encountered the most frequently. For instance, consider how frustrating it would be to have to hit a function key before typing in each character!

The "menu-driven" style of programs mentioned above are almost as bad, since usually only one piece of information is collected at each menu. Such a program becomes a labyrinth of states better suited to an adventure game!

The least offensive modes are found at the higher, goal related levels of the hierarchy. The better they align with changes in the state of the original problem, the more they are tolerated. For example, a word processing program might have one screen layout for program editing, another for writing letters, and yet another while printing the documents. A multi-function business package might have one set of menus for the spreadsheet, another for a graphing module, and a third for a database.

In some cases the problem solved by the program has convenient "fracture lines" which can be used to define the modes. An example in my own past is the RCS, where the editing of each type of resource tree forms its own mode, with each of the modes nested within the overall mode and problem of composing the entire resource tree.

TO DO IS TO BE!

Any narrative description of user interface is bound to be lacking. There is no way text can convey the vibrancy and tactile pleasure of a good interface, or the sullen boredom of a bad one. Therefore, I encourage you to experiment. Get out your favorite arcade game and see if you can spot some of the elements I have described. Dig into your slush pile for the most annoying program you have ever seen, run it and see if you can see mistakes. How would you fix them? Then... go do it to your own program!

AMEN...

This concludes the sermon. I'd like some Feedback as to whether you found this Boring Beyond Belief or Really Hot Stuff. If enough people are interested, homily number two will appear a few episodes from now. The very next installment of ST PRO GEM will go back to basics to explore VDI drawing primitives. In the meantime, you might investigate some of the Good Books on interface design referenced below.

REFERENCES

Stuart K. Card, Thomas P. Moran, and Allen Newell, THE PSYCHOLOGY OF HUMAN-COMPUTER INTERACTION, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.
(Fundamental and indispensible. The volume of experimental results make it weighty. The Good Parts are at the beginning and end.)

"Macintosh User Interface Guidelines", in INSIDE MACINTOSH, Apple Computer, Inc., 1984.
(Yes, Atarians, we have something to learn here. Though not everything "translates", this is a fine piece of principled design work. Read and appreciate.)

James D. Foley, Victor L. Wallace, and Peggy Chan, "The Human Factors of Computer Graphics Interaction Techniques", IEEE Computer Graphics (CG & A), November 1984, pp. 13-48.
(A good overview, including higher level topics which I have postponed to a later article. Excellent bibliography.)

J. D. Foley and A. Van Dam, FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS, Addison Wesley, 1984, Chapters 5 and 6.
(If you can't get the article above, read this. If you are designing graphics apps, buy the whole book! Staggering bibliography.)

Ben Schneidermann, "Direct Manipulation: A Step Beyond Programming Languages", IEEE Computer, August 1983, pp. 57-69.
(What do Pacman and Visicalc have in common? Schneidermann's analysis is vital to creating hot interfaces.)

PART - IX

VDI Graphics: Lines and Solids

This issue of ST PRO GEM is the first in a series of two which will explore the fundamentals of VDI graphics output. In this installment, we will take a look at the commands necessary to output simple graphics such as lines, squares and circles as well as more complex figures such as polygons. The following episode will take a first look at graphics text output, with an emphasis on ways to optimize its drawing speed. It will also include another installment of ONLINE Feedback. As usual, there is a download with this column. You should find it under the name GEMCL9.C in DL3 of ATARI16 (PCS-58).

A BIT OF HISTORY

One of the reasons that the VDI can be confusing is that drawing anything at all, even a simple line, can involve setting around four different VDI parameters before making the draw call! (Given the state of the GEM documents, just FINDING them can be fun!) Looking backwards a bit sheds some light on why the VDI is structured this way, and also gives us a framework for organizing a discussion of graphics output.

The GEM VDI closely follows the so-called GKS standard, which defines capabilities and calling sequences for a standardized graphic input/output system. GKS is itself an evolution from an early system called "Core". Both of these standards were born in the days when pen plotters, vectored graphics displays, and minicomputers were the latest items. So, if you wonder why setting the drawing pen color is a separate command, just think back a few years when it actually meant what it says! (The cynical may choose instead to ponder the benefits of standardization.)

When doing VDI output, it helps if you pretend that the display screen really is a plotter or some other separate device, which has its own internal parameters which you can set up and read back. The class of VDI commands called Attribute Functions let you set the parameters. Output Functions cause the "device" to actually draw someone once it is configured. The Inquire Functions let you read back the parameters if necessary.

There are two parameters which are relevant no matter what type of object you are trying to draw. They are the writing mode and the clipping rectangle. The writing mode is similar to that discussed in the column on raster operations.

It determines what effect the figure you are drawing will have on data already on the screen. The writing mode is set with the call:

```
vswrmode(vdihandle, mode);
```

Vdihandle, here and below, is the handle obtained from grafhandle at the beginning of the program. Mode is a word which may be one of:

1 - Replace Mode
2 - Transparent Mode
3 - XOR mode
4 - Reverse Transparent Mode

In replace mode, whatever is on the screen is overwritten. If you are writing characters, this means the background of each character cell will be erased.

In transparent mode, only the pixels directly under the "positive" part of the image, that is, where 1-bits are being written, will be changed. When writing characters, the background of the cell will be left intact.

In XOR mode, an exclusive or is performed between the screen contents and what is being written. The effect is to reverse the image under areas where a 1-bit occurs.

Reverse transparent is like transparent, but with a "reverse color scheme". That is, only places where a 0-bit is to be put are changed to the current writing color. When you write characters in reverse transparent (over white), the effect is reverse video.

The other common parameter is the clipping rectangle. It defines the area on the screen where the VDI is permitted to draw. Any output which would fall outside of this area is ignored; it is effectively a null operation. The clip rectangle is set with the call:

```
vsclip(vdihandle, flag, pxy);
```

Pxy is a four-word array. Pxy[0] and pxy[1] are the X and Y screen coordinates, respectively, of one corner of your clipping rectangle. Pxy[2] and pxy[3] are the coordinates of the diagonally opposite corner of the rectangle. (When working with the AES, use of a GRECT to define the clip is often more convenient. The routine setclip() in the download does this.)

Flag is set to TRUE if clipping is to be used. If you set it to FALSE, the entire screen is assumed to be fair game.

Normally, you should walk the rectangle list for the current window to obtain your clipping rectangles. (See ST PRO GEM #2 for more details.) However, turning off the clip speeds up all output operations, particularly text. You may do this ONLY when you are absolutely certain that the figure you are drawing will not extend out of the top-most window, or out of a dialog.

THE LINE FORMS ON THE LEFT

The VDI line drawing operations include polyline, arc, elliptical arc, and rounded rectangle. I'll first look at the Attribute Functions for line drawing, then go through the drawing primitives themselves.

The most common used line attributes are color and width. The color is set with:

```
vslcolor(vdihandle, color);
```

where color is one of the standard VDI color indices, ranging from zero to 15. (As discussed in column #6, the color which actually appears will depend on the pallette setting of your ST.)

The line width may only be set to ODD positive values, for reasons of symmetry. If you try to set an even value, the VDI will take the next lower odd value. The call is:

```
vslwidth(vdihandle, width);
```

The two less used line parameters are the end style and pattern. With the end style you can cause the output line to have rounded ends or arrowhead ends. The call is:

```
vslends(vdihandle, beginstyle, endstyle);
```

Beginstyle and endstyle are each words which may have the values zero for square ends (the default), one for arrowed ends, or two for rounded ends. They determine the styles for the starting and finishing ends of the line, respectively.

The line pattern attribute can select dotted or dashed lines as well as more complicated patterns. Before continuing, you should note one warning: VDI line output DOES NOT compensate for pixel aspect ratio. That is, the dashes on a line will look twice as long drawn vertically on a medium-res ST screen as they do when drawn horizontally. The command for setting the pattern is:

```
vsltype(vdihandle, style);
```

Style is a word with a value between 1 and 7. The styles selected are:

1 - Solid (the default)
2 - Long Dash
3 - Dot
4 - Dash, Dot
5 - Dash
6 - Dash, Dot, Dot
7 - (User defined style)

The user defined style is determined by a 16-bit pattern supplied by the application. A one bit in the pattern turns a pixel on, a zero bit leaves it off. The pattern is cycled through repeatedly, using the high bit first. To use a custom style, you must make the call:

```
vsludsty(vdihandle, pattern);
```

before doing vsltype().

As I mentioned above, the line type Output Functions available are polyline, circular and elliptical arc, and rounded rectangle. Each has its own calling sequence. The call for a polyline is:

```
vpline(vdihandle, points, pxy);
```

Points tells how many vertices will appear on the polyline. For instance, a straight line has two vertices: the end and the beginning. A closed square would have five, with the first and last identical. (There is no requirement that the figure described be closed.)

The pxy array contains the X and Y raster coordinates for the vertices, with a total of 2 * points entries. Pxy[0] and pxy[1] are the first X-Y pair, and so on.

If you happen to be using the XOR drawing mode, remember that drawing twice at a point is equivalent to no drawing at all. Therefore, for a figure to appear closed in XOR mode, the final stroke should actually stop one pixel short of the origin of the figure.

You may notice that in the GEM VDI manual the rounded rectangle and arc commands are referred to as GDPs (Generalized Drawing Primitives). This denotation is historical in nature, and has no effect unless you are writing your own VDI bindings.

The rounded rectangle is nice to use for customized buttons in windows and dialogs. It gives a "softer" look to the screen than the standard square objects. The drawing command is:

```
vrbox(vdihandle, pxy);
```

Pxy is a four word array giving opposite corners of the rectangle, just as for the vsclip() call. The corner rounding occurs within the confines of this rectangle. Nothing will protrude unless you specify a line thickness greater than one. The corner rounding is approximately circular; there is no user control over the degree or shape of rounding.

Both the arc and elliptical arc commands use a curious method of specifying angles. The units are tenths of degrees, so an entire circle is 3600 units. The count starts at ninety degrees right of vertical, and proceeds counterclockwise. This means that "3 o'clock" is 0 units, "noon" is 900 units, "9 o'clock" is 1800 units, and 2700 units is at "half-past". 3600 units take you back to "3 o'clock".

The command for drawing a circular arc is:

```
varc(vdihandle, x, y, radius, begin, end);
```

X and y specify the raster coordinates of the center of the circle. Radius specifies the distance from center to all points on the arc. Begin and end are angles given in units as described above, both with values between 0 and 3600. The drawing of the arc ALWAYS proceeds counterclockwise, in the direction of increasing arc number. So values of 0 and 900 for begin and end would draw a quarter circle from "three o'clock" to "noon". Reversing the values would draw the other three quarters of the circle.

A varc() command which specifies a "full turn" is the fastest way to draw a complete circle on the screen. Be warned, however, that the circle drawing algorithm used in the VDI seems to have some serious shortcomings at small radii! You can experiment with the CIRCLE primitive in ST Logo, which uses varc(), to see what I mean.

Notice that if you want an arc to strike one or more given points on the screen, then you are in for some trigonometry. If your math is a bit rusty, I highly recommend the book "A Programmer's Geometry", by Bowyer and Woodwark, published by Butterworths (London, Boston, Toronto).

Finally, the elliptical arc is generated with:

```
vellarc(vdihandle, x, y, xrad, yrad, begin, end);
```

X, y, begin, and end are just as before. Xrad and yrad give the horizontal and vertical radii of the defining ellipse. This means that the distance of the arc from center will be yrad pixels at "noon" and "half-past", and it will be xrad pixels at "3 and 9 o'clock". Again, the arc is always drawn counterclockwise.

There are a number of approaches to keeping the VDI's attributes "in sync" with the actual output operations. Probably the LEAST efficient is to use the Inquire Functions to determine the current attributes. For this reason, I have omitted a discussion of these calls from this column.

Another idea is to keep a local copy of all significant attributes, use a test-before-set method to minimize the number of Attribute Functions which need to be called. This puts a burden on the programmer to be sure that the local attribute variables are correctly maintained. Failure to do so may result in obscure drawing bugs. If your application employs user defined AES objects, you must be very careful because GEM might call your draw code in the middle of a VDI operation (particularly if the user defined objects are in the menu).

Always setting the attributes is a simplistic method, but often proves most effective. The routines plperim() and rrperim() in the download exhibit this approach. Modification for other primitives is straightforward. This style is most useful when drawing operations are scattered throughout the program, so that keeping track of the current attribute status is difficult. Although inherently inefficient, the difference is not very noticable if the drawing operation requested is itself time consuming.

In many applications, such as data graphing programs or "Draw" packages, the output operations are centralized, forming the primary functionality of the code. In this case, it is both easy and efficient to keep track of attribute status between successive drawing operations.

SOLIDS

There are a wider variety of VDI calls for drawing solid figures. They include rectangle or bar, disk, pie, ellipse, elliptical pie, filled rounded rectangle, and

filled polygonal area. Of course, filled figure calls also have their own set of attributes which you will need to set.

The fill color index determines what pen color will be used to draw the solid. It is set with:

```
vsfcolor(vdihandle, color);
```

Color is just the same as for line drawing. A solid may or may not have a visible border. This is determined with the call:

```
vsfperimeter(vdihandle, vis);
```

Vis is a Boolean. If it is true, the figure will be given a solid one pixel outline in the current fill color index. This is often useful to improve the appearance of solids drawn with a dithered fill pattern. If vis is false, then no outline is drawn.

There are two parameters which together determine the pattern used to fill your figure. They are called interior style and interior index. The style determines the general type of fill, and the index is used to select a particular pattern if necessary. The style is set with the command:

```
vsfinterior(vdihandle, style);
```

where style is a value from zero through four. Zero selects a hollow style: the fill is performed in color zero, which is usually white. Style one selects a solid fill with the current fill color. A style of two is called "pattern" and a three is called "hatch", which are terms somewhat suggestive of the options which can then be selected using the interior index. Style four selects the user defined pattern, which is described below.

The interior index is only significant for styles two and three. To set it, use:

```
vsfstyle(vdihandle, index);
```

(Be careful here: it is very easy to confuse this call with the one above due to the unfortunate choice of name.) The index selects the actual drawing pattern. The GEM VDI manual shows fill patterns corresponding to index values from 1 to 24 under style 2, and from 1 to 12 under style 3. However, some of these are implemented differently on the ST. Rather than try to describe them all here, I would suggest that you experiment. You can do so easily in ST Logo by opening the Graphics Settings dialog and playing with the style and index values there.

The user defined style gives you some interesting options for multi-color fills. It is set with:

```
vsfudpat(vdihandle, pattern, planes);
```

Planes determines the number of color planes in the pattern which you supply. It is set to one if you are setting a monochrome pattern. (Remember, monochrome is not necessarily black). It may be set to higher values on color systems: two for ST medium-res mode, or four for low-res mode. If you use a number lower than four under low-res, the other planes are zero filled.

The pattern parameter is an array of words which is a multiple of 16 words long. The pattern determined is 16 by 16 pixels, with each word forming one row of the pattern. The rows are arranged top to bottom, with the most significant bit to the left. If you have selected a multi-plane pattern, the entire first plane is stored, then the second, and so on.

Note that to use a multi-plane pattern, you set the writing mode to replace using vswrmode(). Since the each plane can be different, you can produce multi-colored patterns. If you use a writing color other than black, some of the planes may "disappear".

Most of the solids Output Functions have analogous line drawing commands. The polyline command corresponds to the filled area primitive. The filled area routine is:

```
vfillarea(vdihandle, count, pxy);
```

Count and pxy are just the same as for vpline(). If the polygon defined by pxy is not closed, then the VDI will force closure with a straight line from the last to the first point. The polygon may be concave or self-intersecting. If perimeter show is on, the area will be outlined.

One note of caution is necessary for both vfillarea() and vpline(). There is a limit on the number of points which may be stored in pxy[]. This limit occurs because the contents of pxy[] are copied to the intin[] binding array before the VDI is called. You can determine the maximum number of vertices by checking intout[14] after using the extended inquire function vqextnd().

For reasons unknown to this writer, there are TWO different filled rectangle commands in the VDI. The first is

```
vrrecfl(vdihandle, pxy);
```

Pxy is a four word array defining two opposite corners of the rectangle, just as in vsclip(). Vrrecfl() uses the fill attribute settings, except that it NEVER draws a perimeter.

The other rectangle routine is vbar(), with exactly the same arguments as vrrecfl(). The only difference is that the perimeter setting IS respected. These two routines are the fastest way to produce a solid rectangle using the VDI. They may be used in XOR mode with a BLACK fill color to quickly invert an area of the screen. You can improve the speed even further by turning off the clip (if possible), and byte aligning the left and right edges of the rectangle.

Separate commands are provided for solid circle and ellipse. The circle call is:

```
vcircle(vdihandle, x, y, radius);
```

and the ellipse command is:

```
vellipse(vdihandle, x, y, xrad, yrad);
```

All of the parameters are identical to those given above for varc() and vellarc(). The solid analogue of an arc is a "pie slice". The VDI pie commands are:

```
vpieslice(vdihandle, x, y, radius, begin, end);
```

for a slice from a circular pie, and

```
vellpie(vdihandle, x, y, xrad, yrad, begin, end);
```

for a slice from a "squashed" pie. Again, the parameters are identical to those in varc() and vellarc(). The units and drawing order of angles are also the same. The final solids Output Function is:

```
vrfbox(vdihandle, pxy);
```

which draws a filled rounded rectangle. The pxy array defines two two opposite corners of the bounding box, as shown for vsclip().

The issues involved in correctly setting the VDI attributes for a fill operation are identical to those in drawing lines. For those who want to employ the "always set" method, I have again included two skeleton routines in the download, which can be modified as desired.

TO BE CONTINUED

This concludes the first part of our expedition through basic VDI operations. The next issue will tackle the problems of drawing bit mapped text at a reasonable speed. This first pass will not attempt to tackle alternate or proportional fonts, or alternate font sizes. Instead, I will concentrate on techniques for squeezing greater performance out of the standard monospaced system fonts.

Appendix - I
Sample Code for Part II

```
/* >>>>>>>>>>>>>>>>>>>>>>>> Sample Redraw Code <<<<<<<<<<<<<<<<<<<<<<<<< */

VOID
doredraw(wh, area)                  /* wh = window handle from msg[3] */
WORD    wh;                 /* area = pointer to redraw rect- */
GRECT   *area;              /*   tangle in msg[4] thru msg[7] */
{
        GRECT   box;

        grafmouse(MOFF, 0x0L);
        windupdate(BEGUPDATE);

        windget(wh, WFFIRSTXYWH, &box.gx, &box.gy, &box.gw, &box.gh);
        while ( box.gw && box.gh )
        {
                if (rcintersect(full, &box))      /* Full is entire screen */
                    if (rcintersect(area, &box))
                    {
                            if (wh == w1handle)   /* Test for window 1 handle */
                            {               /* AES redraw example       */
                                objcdraw(w1tree, ROOT, MAXDEPTH, box.gx,
                                        box.gy, box.gw, box.gh);
                            }
                            else if (wh == w2handle) /* Test for window 2 handle */
                            {               /* VDI redraw example       */
                                setclip(TRUE, &box);
                                /*  Put VDI drawing calls here */
                            }
                            /* add more windows here */
                    }
                windget(wh, WFNEXTXYWH, &box.gx, &box.gy, &box.gw,
                        &box.gh);
        }

        windupdate(ENDUPDATE);
        grafmouse(MON, 0x0L);
}
```

```
/* >>>>>>>>>>>>>>>>>>>>>>> Utilities used in doredraw <<<<<<<<<<<<<<<<<<<< */

VOID
setclip(clipflag, area) /* set clip to specified area   */
WORD    clipflag;
GRECT   *area;
{
        WORD    pxy[4];

        grecttoarray(area, pxy);
        vsclip(vdihandle, clipflag, pxy);
}

VOID
grecttoarray(area, array)       /* convert x,y,w,h to upr lt x,y and    */
GRECT   *area;          /*                       lwr rt x,y        */
WORD    *array;
{
        *array++ = area->gx;
        *array++ = area->gy;
        *array++ = area->gx + area->gw - 1;
        *array = area->gy + area->gh - 1;
}

WORD
rcintersect(p1, p2)             /* compute intersect of two rectangles  */
GRECT   *p1, *p2;
{
        WORD    tx, ty, tw, th;

        tw = min(p2->gx + p2->gw, p1->gx + p1->gw);
        th = min(p2->gy + p2->gh, p1->gy + p1->gh);
        tx = max(p2->gx, p1->gx);
        ty = max(p2->gy, p1->gy);
        p2->gx = tx;
        p2->gy = ty;
        p2->gw = tw - tx;
        p2->gh = th - ty;
        return( (tw > tx) && (th > ty) );
}
```

```
/* >>>>>>>>>>>>>>>>>>>>>>> "Self-redraw" Utility <<<<<<<<<<<<<<<<<<<<<<<< */

VOID
sendredraw(wh, p)
WORD    wh;
GRECT   *p;
{
        WORD    msg[8];

        msg[0] = WMREDRAW;              /* Defined in GEMBIND.H     */
        msg[1] = glapid;                /* As returned by applinit */
        msg[2] = 0;
        msg[3] = wh;                    /* Handle of window to redraw */
        msg[4] = p->gx;
        msg[5] = p->gy;
        msg[6] = p->gw;
        msg[7] = p->gh;
        applwrite(glapid, 16, &msg);    /* Use ADDR(msg) for portability */
}
```

```
/* >>>>>>>>>>>>>>>>>>> Utilities for Window Requests <<<<<<<<<<<<<<<<<< */

VOID
rcconstrain(pc, pt)
GRECT           *pc;
GRECT           *pt;
{
        if (pt->gx < pc->gx)
            pt->gx = pc->gx;
        if (pt->gy < pc->gy)
            pt->gy = pc->gy;
        if ((pt->gx + pt->gw) > (pc->gx + pc->gw))
            pt->gx = (pc->gx + pc->gw) - pt->gw;
        if ((pt->gy + pt->gh) > (pc->gy + pc->gh))
            pt->gy = (pc->gy + pc->gh) - pt->gh;
}

WORD
align(x,n)              /* Snap position x to an n-bit grid          */
WORD    x, n;   /* Use n = 16 for horizontal word alignment */
{
        x += (n >> 2) - 1;              /* Round and... */
        x = n * (x / n);                /* remove residue */
        return (x);
}
```

```
/* >>>>>>>>>>>>>>>>>>>>>>> Window full utility <<<<<<<<<<<<<<<<<<<<<<< */

VOID
hndlfull(wh)                /* depending on current window state, make window    */
WORD    wh;         /*   full size -or- return to previous shrunken size */
{                   /* graf calls are optional special effects.          */
        GRECT   prev;
        GRECT   curr;
        GRECT   full;

        windget(wh, WFCXYWH, &curr.gx, &curr.gy, &curr.gw, &curr.gh);
        windget(wh, WFPXYWH, &prev.gx, &prev.gy, &prev.gw, &prev.gh);
        windget(wh, WFFXYWH, &full.gx, &full.gy, &full.gw, &full.gh);
        if ( rcequal(&curr, &full) )
        {               /* Is full, change to previous          */
                grafshrinkbox(prev.gx, prev.gy, prev.gw, prev.gh,
                              full.gx, full.gy, full.gw, full.gh);
                windset(wh, WFCXYWH, prev.gx, prev.gy, prev.gw, prev.gh);
                /* put sendredraw here if you need it */
        }
        else
        {               /* is not full, so set to full          */
                grafgrowbox(curr.gx, curr.gy, curr.gw, curr.gh,
                            full.gx, full.gy, full.gw, full.gh);
                windset(wh, WFCXYWH, full.gx, full.gy, full.gw, full.gh);
        }
}

WORD
rcequal(p1, p2)             /* tests for two rectangles equal       */
GRECT   *p1, *p2;
{
        if ((p1->gx != p2->gx) ||
            (p1->gy != p2->gy) ||
            (p1->gw != p2->gw) ||
            (p1->gh != p2->gh))
            return(FALSE);
        return(TRUE);
}
```

Appendix - II
Sample Code for Part III

```
/*
>>>>>>>>>>>>>>>>>>>>>> Basic Dialog Handler <<<<<<<<<<<<<<<<<<<<<<
*/

WORD
hndldial(tree, def, x, y, w, h)
LONG    tree;
WORD    def;
WORD    x, y, w, h;
{
        WORD    xdial, ydial, wdial, hdial, exitobj;

        formcenter(tree, &xdial, &ydial, &wdial, &hdial);
        formdial(0, x, y, w, h, xdial, ydial, wdial, hdial);
        formdial(1, x, y, w, h, xdial, ydial, wdial, hdial);
        objcdraw(tree, ROOT, MAXDEPTH, xdial, ydial, wdial, hdial);
        exitobj = formdo(tree, def) & 0x7FFF;
        formdial(2, x, y, w, h, xdial, ydial, wdial, hdial);
        formdial(3, x, y, w, h, xdial, ydial, wdial, hdial);
        return (exitobj);
}
```

```
/*
>>>>>>>>>>>>>>>>>>>>>> Object rectangle utility <<<<<<<<<<<<<<<<<<<<<<<<
*/

VOID
objcxywh(tree, obj, p)          /* get x,y,w,h for specified object     */
LONG    tree;
WORD    obj;
GRECT   *p;
{
        objcoffset(tree, obj, &p->gx, &p->gy);
        p->gw = LWGET(OBWIDTH(obj));
        p->gh = LWGET(OBHEIGHT(obj));
}
```

```
/*
>>>>>>>>>>>>>>>>>>>>>> Object flag utilities <<<<<<<<<<<<<<<<<<<<<<<<<
*/

VOID
undoobj(tree, which, bit)       /* clear specified bit in object state  */
LONG    tree;
WORD    which, bit;
{
        WORD    state;

        state = LWGET(OBSTATE(which));
        LWSET(OBSTATE(which), state & ~bit);
}

VOID
deselobj(tree, which)           /* turn off selected bit of spcfd object*/
LONG    tree;
WORD    which;
{
        undoobj(tree, which, SELECTED);
}

VOID
doobj(tree, which, bit) /* set specified bit in object state    */
LONG    tree;
WORD    which, bit;
{
        WORD    state;

        state = LWGET(OBSTATE(which));
        LWSET(OBSTATE(which), state | bit);
}

VOID
selobj(tree, which)             /* turn on selected bit of spcfd object */
LONG    tree;
WORD    which;
{
        doobj(tree, which, SELECTED);
}

BOOLEAN
statep(tree, which, bit)
LONG    tree;
WORD    which;
WORD    bit;
{
        return ( (LWGET(OBSTATE(which)) & bit) != 0);
}

BOOLEAN
```

```
selectp(tree, which)
LONG    tree;
WORD    which;
{
        return statep(tree, which, SELECTED);
}
```

```
/*
>>>>>>>>>>>>>>>>>>>>> Sample radio buttons after dialog <<<<<<<<<<<<<<<<<<<<
*/


WORD
encode(tree, ob1st, num)
LONG    tree;
WORD    ob1st, num;
{
        for (; num--; )
                if (selectp(ob1st+num))
                        return(num);
        return (-1);
}
```

Appendix - III
Sample Code for Part IV

```
/*
>>>>>>>>>>>>>>>>>>>>>>>> Sample C output file from RCS <<<<<<<<<<<<<<<<<<<<
*/
                                        /* (Comments added)     */
BYTE *rsstrings[] = {                   /* ASCII data           */
"Title String",
"Exit",
"Centered Text",
"",
"",
"Butt",
"Tokyo",
"",
"Time: _:_:_",
"999999",
"",
"Time: _:_:_  ",
"999999",
"New York"};

WORD IMAG0[] = {                                /* Bitmap for GIMAGE */
0x7FF, 0xFFFF, 0xFF80, 0xC00,
0x0, 0xC0, 0x183F, 0xF03F,
0xF060, 0x187F, 0xF860, 0x1860,
0x187F, 0xF860, 0x1860, 0x187F,
0xF860, 0x1860, 0x187F, 0xF860,
0x1860, 0x187F, 0xF860, 0x1860,
0x187F, 0xF860, 0x1860, 0x187F,
0xF860, 0x1860, 0x187F, 0xF860,
0x1860, 0x187F, 0xF860, 0x1860,
0x187F, 0xF860, 0x1860, 0x187F,
0xF860, 0x1860, 0x183F, 0xF03F,
0xF060, 0xC00, 0x0, 0xC0,
0x7FF, 0xFFFF, 0xFF80, 0x0,
0x0, 0x0, 0x3F30, 0xC787,
0x8FE0, 0xC39, 0xCCCC, 0xCC00,
0xC36, 0xCFCC, 0xF80, 0xC30,
0xCCCD, 0xCC00, 0x3F30, 0xCCC7,
0xCFE0, 0x0, 0x0, 0x0};

WORD IMAG1[] = {                                /* Mask for first icon */
0x0, 0x0, 0x0, 0x0,
0x7FFE, 0x0, 0x1F, 0xFFFF,
0xFC00, 0xFF, 0xFFFF, 0xFF00,
0x3FF, 0xFFFF, 0xFFC0, 0xFFF,
0xFFFF, 0xFFF0, 0x3FFF, 0xFFFF,
0xFFFC, 0x7FFF, 0xFFFF, 0xFFFE,
```

```
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0x7FFF,
0xFFFF, 0xFFFE, 0x3FFF, 0xFFFF,
0xFFFC, 0xFFF, 0xFFFF, 0xFFF0,
0x3FF, 0xFFFF, 0xFFC0, 0xFF,
0xFFFF, 0xFF00, 0x1F, 0xFFFF,
0xF800, 0x0, 0x7FFE, 0x0};

WORD IMAG2[] = {                                /* Data for first icon */
0x0, 0x0, 0x0, 0x0,
0x3FFC, 0x0, 0xF, 0xC003,
0xF000, 0x78, 0x180, 0x1E00,
0x180, 0x180, 0x180, 0x603,
0x180, 0xC060, 0x1C00, 0x6,
0x38, 0x3000, 0x18C, 0xC,
0x60C0, 0x198, 0x306, 0x6000,
0x1B0, 0x6, 0x4000, 0x1E0,
0x2, 0xC000, 0x1C0, 0x3,
0xCFC0, 0x180, 0x3F3, 0xC000,
0x0, 0x3, 0x4000, 0x0,
0x2, 0x6000, 0x0, 0x6,
0x60C0, 0x0, 0x306, 0x3000,
0x0, 0xC, 0x1C00, 0x0,
0x38, 0x603, 0x180, 0xC060,
0x180, 0x180, 0x180, 0x78,
0x180, 0x1E00, 0xF, 0xC003,
0xF000, 0x0, 0x3FFC, 0x0};

WORD IMAG3[] = {                                /* Mask for second icon */
0x0, 0x0, 0x0, 0x0,
0x7FFE, 0x0, 0x1F, 0xFFFF,
0xFC00, 0xFF, 0xFFFF, 0xFF00,
0x3FF, 0xFFFF, 0xFFC0, 0xFFF,
0xFFFF, 0xFFF0, 0x3FFF, 0xFFFF,
0xFFFC, 0x7FFF, 0xFFFF, 0xFFFE,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0x7FFF,
0xFFFF, 0xFFFE, 0x3FFF, 0xFFFF,
0xFFFC, 0xFFF, 0xFFFF, 0xFFF0,
0x3FF, 0xFFFF, 0xFFC0, 0xFF,
0xFFFF, 0xFF00, 0x1F, 0xFFFF,
0xF800, 0x0, 0x7FFE, 0x0};
```

```
WORD IMAG4[] = {                        /* Data for second icon */
0x0, 0x0, 0x0, 0x0,
0x3FFC, 0x0, 0xF, 0xC003,
0xF000, 0x78, 0x180, 0x1E00,
0x180, 0x180, 0x180, 0x603,
0x180, 0xC060, 0x1C00, 0x6,
0x38, 0x3000, 0x18C, 0xC,
0x60C0, 0x198, 0x306, 0x6000,
0x1B0, 0x6, 0x4000, 0x1E0,
0x2, 0xC000, 0x1C0, 0x3,
0xCFC0, 0x180, 0x3F3, 0xC000,
0x0, 0x3, 0x4000, 0x0,
0x2, 0x6000, 0x0, 0x6,
0x60C0, 0x0, 0x306, 0x3000,
0x0, 0xC, 0x1C00, 0x0,
0x38, 0x603, 0x180, 0xC060,
0x180, 0x180, 0x180, 0x78,
0x180, 0x1E00, 0xF, 0xC003,
0xF000, 0x0, 0x3FFC, 0x0};

LONG rsfrstr[] = {                      /* Free string index - unused */
0};

BITBLK rsbitblk[] = {                   /* First entry is index to image data */
0L, 6, 24, 0, 0, 0};

LONG rsfrimg[] = {                      /* Free image index - unused */
0};

ICONBLK rsiconblk[] = {
1L, 2L, 10L, 4096,0,0, 0,0,48,24, 9,24,30,8,    /* First pointer is mask */
3L, 4L, 17L, 4864,0,0, 0,0,48,24, 0,24,48,8};   /* Second is data, third */
                                                /* is to title string    */
TEDINFO rstedinfo[] = {
2L, 3L, 4L, 3, 6, 2, 0x1180, 0x0, -1, 14,1,     /* First pointer is text */
7L, 8L, 9L, 3, 6, 2, 0x2072, 0x0, -3, 11,1,     /* Second is template    */
11L, 12L, 13L, 3, 6, 0, 0x1180, 0x0, -1, 1,15,  /* Third is validation   */
14L, 15L, 16L, 3, 6, 1, 0x1173, 0x0, 0, 1,17};

OBJECT rsobject[] = {
-1, 1, 3, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 18,12,   /* Pointers are to: */
2, -1, -1, GSTRING, NONE, NORMAL, 0x0L, 3,1, 12,1,      /* rsstrings        */
3, -1, -1, GBUTTON, 0x7, NORMAL, 0x1L, 5,9, 8,1,        /* rsstrings        */
0, 4, 4, GBOX, NONE, NORMAL, 0xFF1172L, 3,3, 12,5,
3, -1, -1, GIMAGE, LASTOB, NORMAL, 0x0L, 3,1, 6,3,      /* rsbitblk         */
-1, 1, 6, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 23,12,
2, -1, -1, GTEXT, NONE, NORMAL, 0x0L, 0,1, 23,1,        /* rstedinfo        */
6, 3, 5, GIBOX, NONE, NORMAL, 0x1100L, 6,3, 11,5,
4, -1, -1, GBUTTON, 0x11, NORMAL, 0x5L, 0,0, 11,1,      /* rsstrings        */
5, -1, -1, GBUTTON, 0x11, NORMAL, 0x6L, 0,2, 11,1,      /* rsstrings        */
2, -1, -1, GBOXCHAR, 0x11, NORMAL, 0x43FF1400L, 0,4, 11,1,
```

```
0, -1, -1, GBOXTEXT, 0x27, NORMAL, 0x1L, 5,9, 13,1,     /* rstedinfo    */
-1, 1, 3, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 32,11,
2, -1, -1, GICON, NONE, NORMAL, 0x0L, 4,1, 6,4, /* rsiconblk    */
3, -1, -1, GFTEXT, EDITABLE, NORMAL, 0x2L, 12,2, 14,1,  /* rstedinfo    */
0, 4, 4, GFBOXTEXT, 0xE, NORMAL, 0x3L, 3,5, 25,4,       /* rstedinfo    */
3, -1, -1, GICON, LASTOB, NORMAL, 0x1L, 1,0, 6,4};      /* rsiconblk    */

LONG rstrindex[] = {                        /* Points to start of trees in */
0L,                                         /* rsobject                    */
5L,
12L};

struct foobar {                             /* Temporary structure used by    */
        WORD    dummy;                      /* RSCREATE when setting up image */
        WORD    *image;                     /* pointers.                      */
        } rsimdope[] = {
0, &IMAG0[0],
0, &IMAG1[0],
0, &IMAG2[0],
0, &IMAG3[0],
0, &IMAG4[0]};

                                            /* Counts of structures defined */
#define NUMSTRINGS 18
#define NUMFRSTR 0
#define NUMIMAGES 5
#define NUMBB 1
#define NUMFRIMG 0
#define NUMIB 2
#define NUMTI 4
#define NUMOBS 17
#define NUMTREE 3

BYTE pname[] = "DEMO.RSC";
```

```
/*
>>>>>>>>>>>>>>>>>>>>>>>>>>>> Title change utility <<<<<<<<<<<<<<<<<<<<<<<<
*/
        VOID
settext(tree, obj, str)
        LONG    tree, str;
        WORD    obj;
        {
        LONG    obspec;

        obspec = LLGET(OBSPEC(obj));            /* Get TEDINFO address  */
        LLSET(TEPTEXT(obspec), str);            /* Set new text pointer */
        LWSET(TETXTLEN(obspec), LSTRLEN(str)); /* Set new length       */
        }
```

```
/*
>>>>>>>>>>>>>>>>>>>>>> Text edit code segment <<<<<<<<<<<<<<<<<<<<<<<<<
*/
        LONG    tree, obspec;
        BYTE    text[41];

        rsrcgaddr(RTREE, DIALOG, &tree);        /* Get tree address  */
        obspec = LLGET(OBSPEC(EDITOBJ));        /* Get TEDINFO address  */
        LLSET(TEPTEXT(obspec), ADDR(str));      /* Set new text pointer */
        LWSET(TETXTLEN(obspec), 41);            /* Set max length       */
        text[0] = '\0';                         /* Make empty string */
```

```
/*
>>>>>>>>>>>>>>>>>>>> Sample 68K only source code <<<<<<<<<<<<<<<<<<<<<<
*/
        VOID
settext(tree, obj, str)
        OBJECT  *tree;
        WORD    obj;
        BYTE    *str;
        {
        TEDINFO *obspec;

        obspec = (TEDINFO *) (tree + obj)->obspec;
                                                /* Get TEDINFO address  */
        obspec->teptext = str;                  /* Set new text pointer */
        obspec->tetxtlen = strlen(str); /* Set new length       */
        }
```

```
/*
>>>>>>>>>>>>>>>>>>>>>>>>>>>> Symbol definitions <<<<<<<<<<<<<<<<<<<<<<<<<<
*/
                                                    /* Window parts */
#define NAME 0x0001
#define CLOSER 0x0002
#define FULLER 0x0004
#define MOVER 0x0008
#define INFO 0x0010
#define SIZER 0x0020
#define UPARROW 0x0040
#define DNARROW 0x0080
#define VSLIDE 0x0100
#define LFARROW 0x0200
#define RTARROW 0x0400
#define HSLIDE 0x0800

#define WFKIND 1                                    /* windget/set parameters */
#define WFNAME 2
#define WFINFO 3
#define WFWXYWH 4
#define WFCXYWH 5
#define WFPXYWH 6
#define WFFXYWH 7
#define WFHSLIDE 8
#define WFVSLIDE 9
#define WFTOP 10
#define WFFIRSTXYWH 11
#define WFNEXTXYWH 12
#define WFNEWDESK 14
#define WFHSLSIZ 15
#define WFVSLSIZ 16
                                                    /* window messages      */
#define WMREDRAW 20
#define WMTOPPED 21
#define WMCLOSED 22
#define WMFULLED 23
#define WMARROWED 24
#define WMHSLID 25
#define WMVSLID 26
#define WMSIZED 27
#define WMMOVED 28
#define WMNEWTOP 29
                                                    /* arrow messages       */
#define WAUPPAGE 0
#define WADNPAGE 1
#define WAUPLINE 2
#define WADNLINE 3
#define WALFPAGE 4
#define WARTPAGE 5
#define WALFLINE 6
#define WARTLINE 7
```

```
#define RTREE 0                                 /* Redraw definitions   */
#define ROOT 0
#define MAXDEPTH 8

                                                /* update flags         */
#define ENDUPDATE 0
#define BEGUPDATE 1
#define ENDMCTRL  2
#define BEGMCTRL  3

                                                /* Mouse state changes  */
#define MOFF 256
#define MON 257

                                                /* Object flags         */
#define NONE        0x0
#define SELECTABLE  0x1
#define DEFAULT     0x2
#define EXIT        0x4
#define EDITABLE    0x8
#define RBUTTON    0x10

                                                /* Object states        */
#define SELECTED   0x1
#define CROSSED    0x2
#define CHECKED    0x4
#define DISABLED   0x8
#define OUTLINED  0x10
#define SHADOWED  0x20

#define GBOX      20
#define GTEXT     21
#define GBOXTEXT  22
#define GIMAGE    23
#define GIBOX     25
#define GBUTTON   26
#define GBOXCHAR  27
#define GSTRING   28
#define GFTEXT    29
#define GFBOXTEXT 30
#define GICON     31
#define GTITLE    32

                                                /* Data structures      */
typedef struct grect
        {
        int     gx;
        int     gy;
        int     gw;
        int     gh;
        } GRECT;

typedef struct object
        {
        int             obnext; /* -> object's next sibling     */
        int             obhead; /* -> head of object's children */
```

```
        int             obtail; /* -> tail of object's children */
        unsigned int    obtype; /* type of object- BOX, CHAR,...*/
        unsigned int    obflags;        /* flags                        */
        unsigned int    obstate;        /* state- SELECTED, OPEN, ...   */
        long            obspec; /* "out"- -> anything else      */
        int             obx;            /* upper left corner of object  */
        int             oby;            /* upper left corner of object  */
        int             obwidth;        /* width of obj                 */
        int             obheight;       /* height of obj                */
        } OBJECT;

typedef struct textedinfo
        {
        long            teptext;        /* ptr to text (must be 1st)    */
        long            teptmplt;       /* ptr to template              */
        long            tepvalid;       /* ptr to validation chrs.      */
        int             tefont; /* font                                 */
        int             tejunk1;        /* junk word                    */
        int             tejust; /* justification- left, right...*/
        int             tecolor;        /* color information word       */
        int             tejunk2;        /* junk word                    */
        int             tethickness;    /* border thickness             */
        int             tetxtlen;       /* length of text string        */
        int             tetmplen;       /* length of template string    */
        } TEDINFO;

                                        /* "Portable" data definitions */
#define OBNEXT(x) (tree + (x) * sizeof(OBJECT) + 0)
#define OBHEAD(x) (tree + (x) * sizeof(OBJECT) + 2)
#define OBTAIL(x) (tree + (x) * sizeof(OBJECT) + 4)
#define OBTYPE(x) (tree + (x) * sizeof(OBJECT) + 6)
#define OBFLAGS(x) (tree + (x) * sizeof(OBJECT) + 8)
#define OBSTATE(x) (tree + (x) * sizeof(OBJECT) + 10)
#define OBSPEC(x) (tree + (x) * sizeof(OBJECT) + 12)
#define OBX(x) (tree + (x) * sizeof(OBJECT) + 16)
#define OBY(x) (tree + (x) * sizeof(OBJECT) + 18)
#define OBWIDTH(x) (tree + (x) * sizeof(OBJECT) + 20)
#define OBHEIGHT(x) (tree + (x) * sizeof(OBJECT) + 22)

#define TEPTEXT(x)  (x)
#define TETXTLEN(x)  (x + 24)
```

Appendix - IV
Sample Code for Part V

```
/*
>>>>>>>>>>>>>>>>>>>>>>>>> Sample object trees <<<<<<<<<<<<<<<<<<<<<<<<
*/

OBJECT rsobject[] = {
-1, 1, 3, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 18,12,    /* Tree # 1 */
2, -1, -1, GSTRING, NONE, NORMAL, 0x0L, 3,1, 12,1,
3, -1, -1, GBUTTON, 0x7, NORMAL, 0x1L, 5,9, 8,1,
0, 4, 4, GBOX, NONE, NORMAL, 0xFF1172L, 3,3, 12,5,
3, -1, -1, GIMAGE, LASTOB, NORMAL, 0x0L, 3,1, 6,3,
-1, 1, 6, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 23,12,  /* Tree # 2 */
2, -1, -1, GTEXT, NONE, NORMAL, 0x0L, 0,1, 23,1,
6, 3, 5, GIBOX, NONE, NORMAL, 0x1100L, 6,3, 11,5,
4, -1, -1, GBUTTON, 0x11, NORMAL, 0x5L, 0,0, 11,1,
5, -1, -1, GBUTTON, 0x11, NORMAL, 0x6L, 0,2, 11,1,
2, -1, -1, GBOXCHAR, 0x11, NORMAL, 0x43FF1400L, 0,4, 11,1,
0, -1, -1, GBOXTEXT, 0x27, NORMAL, 0x1L, 5,9, 13,1,
-1, 1, 3, GBOX, NONE, OUTLINED, 0x21100L, 0,0, 32,11,  /* Tree # 3 */
2, -1, -1, GICON, NONE, NORMAL, 0x0L, 4,1, 6,4,
3, -1, -1, GFTEXT, EDITABLE, NORMAL, 0x2L, 12,2, 14,1,
0, 4, 4, GFBOXTEXT, 0xE, NORMAL, 0x3L, 3,5, 25,4,
3, -1, -1, GICON, LASTOB, NORMAL, 0x1L, 1,0, 6,4};
```

```
/*
>>>>>>>>>>>>>>>>>>>>>>>> Object tree walk utility <<<<<<<<<<<<<<<<<<<<<<<<
*/
        VOID
maptree(tree, this, last, routine)
        LONG            tree;
        WORD            this, last;
        WORD            (*routine)();
        {
        WORD            tmp1;

        tmp1 = this;                    /* Initialize to impossible value: */
                                        /* TAIL won't point to self!       */
                                        /* Look until final node, or off   */
                                        /* the end of tree                 */
        while (this != last && this != NIL)
                                        /* Did we 'pop' into this node     */
                                        /* for the second time?            */
                if (LWGET(OBTAIL(this)) != tmp1)
                        {
                        tmp1 = this;    /* This is a new node       */
                        this = NIL;
                                        /* Apply operation, testing  */
                                        /* for rejection of sub-tree */
                        if ((*routine)(tree, tmp1))
                                this = LWGET(OBHEAD(tmp1));
                                        /* Subtree path not taken,   */
                                        /* so traverse right         */
                        if (this == NIL)
                                this = LWGET(OBNEXT(tmp1));
                        }
                else                    /* Revisiting parent:        */
                                        /* No operation, move right  */
                        {
                        tmp1 = this;
                        this = LWGET(OBNEXT(tmp1));
                        }
        }
```

```
/*
>>>>>>>>>>>>>>>>>> Sample routine to use with maptree() <<<<<<<<<<<<<<<
*/
        VOID
undoobj(tree, which, bit)       /* clear specified bit in object state  */
        LONG    tree;
        WORD    which, bit;
        {
        WORD    state;

        state = LWGET(OBSTATE(which));
        LWSET(OBSTATE(which), state & ~bit);
        }

        VOID
deselobj(tree, which)           /* turn off selected bit of spcfd object*/
        LONG    tree;
        WORD    which;
        {
        undoobj(tree, which, SELECTED);
        return (TRUE);
        }
```

```
/*
>>>>>>>>>>>>>>>>>>>>>>>> Sample .ICN Files <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>> Save everything between >>><<< lines as CLOCK.ICN <<<<<<<<<<<<<<
*/

/* GEM Icon Definition: */
#define ICONW 0x0030
#define ICONH 0x0018
#define DATASIZE 0x0048
UWORD clock[DATASIZE] =
{ 0x0000, 0x0000, 0x0000, 0x0000,
  0x3FFC, 0x0000, 0x000F, 0xC003,
  0xF000, 0x0078, 0x0180, 0x1E00,
  0x0180, 0x0180, 0x0180, 0x0603,
  0x0180, 0xC060, 0x1C00, 0x0006,
  0x0038, 0x3000, 0x018C, 0x000C,
  0x60C0, 0x0198, 0x0306, 0x6000,
  0x01B0, 0x0006, 0x4000, 0x01E0,
  0x0002, 0xC000, 0x01C0, 0x0003,
  0xCFC0, 0x0180, 0x03F3, 0xC000,
  0x0000, 0x0003, 0x4000, 0x0000,
  0x0002, 0x6000, 0x0000, 0x0006,
  0x60C0, 0x0000, 0x0306, 0x3000,
  0x0000, 0x000C, 0x1C00, 0x0000,
  0x0038, 0x0603, 0x0180, 0xC060,
  0x0180, 0x0180, 0x0180, 0x0078,
  0x0180, 0x1E00, 0x000F, 0xC003,
  0xF000, 0x0000, 0x3FFC, 0x0000
};
/*
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> End of CLOCK.ICN <<<<<<<<<<<<<<<<<<<<<<<<<<
*/
```

```
/*
>>>>>>>>> Save everything between >>>>>>>>> lines as CLOCKM.ICN <<<<<<<<<<
*/

/* GEM Icon Definition: */
#define ICONW 0x0030
#define ICONH 0x0018
#define DATASIZE 0x0048
UWORD clockm[DATASIZE] =
{ 0x0000, 0x0000, 0x0000, 0x0000,
  0x7FFE, 0x0000, 0x001F, 0xFFFF,
  0xFC00, 0x00FF, 0xFFFF, 0xFF00,
  0x03FF, 0xFFFF, 0xFFC0, 0x0FFF,
  0xFFFF, 0xFFF0, 0x3FFF, 0xFFFF,
  0xFFFC, 0x7FFF, 0xFFFF, 0xFFFE,
  0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF, 0xFFFF, 0x7FFF,
  0xFFFF, 0xFFFE, 0x3FFF, 0xFFFF,
  0xFFFC, 0x0FFF, 0xFFFF, 0xFFF0,
  0x03FF, 0xFFFF, 0xFFC0, 0x00FF,
  0xFFFF, 0xFF00, 0x001F, 0xFFFF,
  0xF800, 0x0000, 0x7FFE, 0x0000
};
/*
>>>>>>>>>>>>>>>>>>>>>>>>> End of CLOCKM.ICN <<<<<<<<<<<<<<<<<<<<<<<<<<<<
*/
```

Appendix - V
Sample Code for Part VI

```
/* >>>>>>>>>>>>>>>>>>>>>>>>>>> MFDB Structure <<<<<<<<<<<<<<<<<<<<<<<<<< */
                                        /* Memory Form Definition Block */
typedef struct fdbstr
{
        long            fdaddr; /* Form address                      */
        int             fdw;            /* Form width in pixels          */
        int             fdh;            /* Form height in pixels         */
        int             fdwdwidth;      /* Form width in memory words    */
        int             fdstand;        /* Standard form flag            */
        int             fdnplanes;      /* Number of color planes        */
        int             fdr1;           /* Dummy locations:              */
        int             fdr2;           /* Reserved for future use       */
        int             fdr3;
} MFDB;
```

```
/* >>>>>>>>>>>>>>>>>>>> Resource Transform Utilities <<<<<<<<<<<<<<<<<< */
/*-------------------------------*/
/*      vdifix                   */
/*-------------------------------*/
VOID
vdifix(pfd, theaddr, wb, h)             /* This routine loads the MFDB */
        MFDB            *pfd;           /* Input values are the MFDB's */
        LONG            theaddr;        /* address, the form's address,*/
        WORD            wb, h;          /* the form's width in bytes,  */
        {                               /* and the height in pixels    */
        pfd->fww = wb >> 1;
        pfd->fwp = wb << 3;
        pfd->fh = h;
        pfd->np = 1;                    /* Monochrome assumed          */
        pfd->mp = theaddr;
        }

/*-------------------------------*/
/*      vditrans                 */
/*-------------------------------*/
WORD
vditrans(saddr, swb, daddr, dwb, h)     /* Transform the standard form */
        LONG            saddr;          /* pointed at by saddr and     */
        UWORD           swb;            /* store in the form at daddr  */
        LONG            daddr;          /* Byte widths and pixel height*/
        UWORD           dwb;            /* are given                   */
        UWORD           h;
        {
        MFDB            src, dst;       /* These are on-the-fly MFDBs  */

        vdifix(&src, saddr, swb, h);    /* Load the source MFDB        */
        src.ff = TRUE;                  /* Set it's std form flag      */

        vdifix(&dst, daddr, dwb, h);    /* Load the destination MFDB   */
        dst.ff = FALSE;                 /* Clear the std flag          */
        vrtrnfm(vdihandle, &src, &dst );        /* Call the VDI        */
        }

/*-------------------------------*/
/*      transbitblk              */
/*-------------------------------*/
VOID
transbitblk(obspec)                     /* Transform the image belonging */
        LONG    obspec;                 /* to the bitblk pointed to by   */
        {                               /* obspec.  This routine may also*/
        LONG    taddr;                  /* be used with free images      */
        WORD    wb, hl;

        if ( (taddr = LLGET(BIPDATA(obspec))) == -1L)
                return;                 /* Get and validate image address */
        wb = LWGET(BIWB(obspec));       /* Extract image dimensions       */
        hl = LWGET(BIHL(obspec));
```

```
        vditrans(taddr, wb, taddr, wb, hl);     /* Perform a transform    */
        }                                       /* in place               */

/*------------------------------*/
/*      transobj                */
/*------------------------------*/
VOID
transobj(tree, obj)                         /* Examine the input object.  If  */
        LONG    tree;                       /* it is an icon or image, trans- */
        WORD    obj;                        /* form the associated raster     */
        {                                   /* forms in place.                */
        WORD    type, wb, hl;               /* This routine may be used with  */
        LONG    taddr, obspec;              /* maptree() to transform an      */
                                            /* entire resource tree           */

        type = LLOBT(LWGET(OBTYPE(obj)));                   /* Load object type */
        if ( (obspec = LLGET(OBSPEC(obj))) == -1L)          /* Load and check   */
                return (TRUE);                              /* obspec pointer   */
        switch (type) {
                case GIMAGE:
                        transbitblk(obspec);                /* Transform image  */
                        return (TRUE);
                case GICON:                                 /* Load icon size   */
                        hl = LWGET(IBHICON(obspec));
                        wb = (LWGET(IBWICON(obspec)) + 7) >> 3;
                                                            /* Transform data   */
                        if ( (taddr = LLGET(IBPDATA(obspec))) != -1L)
                                vditrans(taddr, wb, taddr, wb, hl);
                                                            /* Transform mask   */
                        if ( (taddr = LLGET(IBPMASK(obspec))) != -1L)
                                vditrans(taddr, wb, taddr, wb, hl);
                        return (TRUE);
                default:
                        return (TRUE);
                }
        }
```

```
/* >>>>>>>>>>>>>>>>  Macro definitions for the code above <<<<<<<<<<<<<<<< */

#define BIPDATA(x)      (x)
#define BIWB(x) (x + 4)
#define BIHL(x) (x + 6)
#define OBTYPE(x)       (tree + (x) * sizeof(OBJECT) + 6)
#define OBSPEC(x)       (tree + (x) * sizeof(OBJECT) + 12)
#define IBPMASK(x)      (x)
#define IBPDATA(x)      (x + 4)
#define IBWICON(x)      (x + 22)
#define IBHICON(x)      (x + 24)
```

```
/* >>>>>>>>>>>>>>>>>>>>>>>> VDI Copy Mode Table <<<<<<<<<<<<<<<<<<<<<<<<< */

Symbols: N = new destination pixel value (0 or 1)
         D = old destination pixel value (0 or 1)
         S = source pixel value (0 or 1)
         ~ = Boolean not (inversion)
         & = Boolean and
         | = Boolean or
         ^ = Boolean xor (exclusive-or)

Mode Number     Action
-----------     ------
    0           N = 0           (USE VBAR INSTEAD)
    1           N = S & D
    2           N = S & ~D
    3           N = S           (REPLACE)
    4           N = ~S & D      (ERASE)
    5           N = D           (USELESS)
    6           N = S ^ D       (XOR)
    7           N = S | D       (TRANSPARENT)
    8           N = ~ (S | D)
    9           N = ~ (S ^ D)
   10           N = ~D          (USE VBAR INSTEAD)
   11           N = S | ~D
   12           N = ~S
   13           N = ~S | D      (REVERSE TRANSPARENT)
   14           N = ~ (S & D)
   15           N = 1           (USE VBAR INSTEAD)

/* >>>>>>>>>>>>>>>>>>>>>>>> END OF DOWNLOAD <<<<<<<<<<<<<<<<<<<<<<<<< */
```

Appendix - VI
Sample Code for Part VII

```
/*
>>>>>>>>>>>>>>>>>>>>>> Download file for GEM column #7 <<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>> Sample Menu Tree <<<<<<<<<<<<<<<<<<<<<<<<<<<<
*/

-1, 1, 6, GIBOX, NONE, NORMAL, 0x0L, 0,0, 80,25,          /* ROOT */
6, 2, 2, GBOX, NONE, NORMAL, 0x1100L, 0,0, 80,513,        /* THE BAR */
1, 3, 5, GIBOX, NONE, NORMAL, 0x0L, 2,0, 20,769,          /* THE ACTIVE */
4, -1, -1, GTITLE, NONE, NORMAL, 0x0L, 0,0, 6,769,        /* Title #1 */
5, -1, -1, GTITLE, NONE, NORMAL, 0x1L, 6,0, 6,769,        /* Title #2 */
2, -1, -1, GTITLE, NONE, NORMAL, 0x2L, 12,0, 8,769,       /* Title #3 */
0, 7, 22, GIBOX, NONE, NORMAL, 0x0L, 0,769, 80,19,        /* THE SCREEN */
16, 8, 15, GBOX, NONE, NORMAL, 0xFF1100L, 2,0, 20,8,      /* Drop-down #1 */
9, -1, -1, GSTRING, NONE, NORMAL, 0x3L, 0,0, 19,1,        /* About... entry */
10, -1, -1, GSTRING, NONE, DISABLED, 0x4L, 0,1, 20,1,
11, -1, -1, GSTRING, NONE, NORMAL, 0x5L, 0,2, 20,1,       /* Desk acc entries */
12, -1, -1, GSTRING, NONE, NORMAL, 0x6L, 0,3, 20,1,
13, -1, -1, GSTRING, NONE, NORMAL, 0x7L, 0,4, 20,1,
14, -1, -1, GSTRING, NONE, NORMAL, 0x8L, 0,5, 20,1,
15, -1, -1, GSTRING, NONE, NORMAL, 0x9L, 0,6, 20,1,
7, -1, -1, GSTRING, NONE, NORMAL, 0xAL, 0,7, 20,1,
22, 17, 21, GBOX, NONE, NORMAL, 0xFF1100L, 8,0, 13,5,     /* Drop-down #2 */
18, -1, -1, GSTRING, NONE, NORMAL, 0xBL, 0,0, 13,1,
19, -1, -1, GSTRING, NONE, DISABLED, 0xCL, 0,1, 13,1,
20, -1, -1, GSTRING, NONE, NORMAL, 0xDL, 0,4, 13,1,
21, -1, -1, GSTRING, NONE, NORMAL, 0xEL, 0,2, 13,1,
16, -1, -1, GSTRING, NONE, DISABLED, 0xFL, 0,3, 13,1,
6, 23, 25, GBOX, NONE, NORMAL, 0xFF1100L, 14,0, 26,3,    /* Drop down #3 */
24, -1, -1, GSTRING, NONE, NORMAL, 0x10L, 0,2, 26,1,
25, -1, -1, GSTRING, NONE, NORMAL, 0x11L, 0,0, 26,1,
22, -1, -1, GSTRING, LASTOB, DISABLED, 0x12L, 0,1, 26,1
```

```
/*
>>>>>>>>>>>>>>>>>>>>>>> Menu enable/disable utility <<<<<<<<<<<<<<<<<<<<<<<
*/

/*-------------------------------*/
/*      undoobj          */
/*-------------------------------*/
        VOID
undoobj(tree, which, bit)
        LONG    tree;
        WORD    which;
        UWORD   bit;
        {
        WORD    state;

        state = LWGET(OBSTATE(which));
        LWSET(OBSTATE(which), state & ~bit);
        }

/*-------------------------------*/
/*      enabobj          */
/*-------------------------------*/
        WORD
enabobj(tree, which)
        LONG    tree;
        WORD    which;
        {
        undoobj(tree, which, (UWORD) DISABLED);
        return (TRUE);
        }

/*-------------------------------*/
/*      doobj            */
/*-------------------------------*/
        VOID
doobj(tree, which, bit)
        LONG    tree;
        WORD    which;
        UWORD   bit;
        {
        WORD    state;

        state = LWGET(OBSTATE(which));
        LWSET(OBSTATE(which), state | bit);
        }

/*-------------------------------*/
/*      disabobj                 */
/*-------------------------------*/
        WORD
disabobj(tree, which)
        LONG    tree;
```

```
        WORD    which;
        {
        doobj(tree, which, (UWORD) DISABLED);
        return (TRUE);
        }

/*------------------------------*/
/*      setmenu         */
/*------------------------------*/
        VOID
setmenu(tree, change)                   /* change[0] TRUE selects all entries*/
        LONG    tree;                   /* FALSE deselects all.  Change list */
        WORD    *change;                /* of items is then toggled.         */
        {
        WORD    dflt, screen, drop, obj;

        dflt = *change++;                       /* What is default?   */
        screen = LWGET(OBTAIL(ROOT));           /* Get SCREEN         */
        drop = LWGET(OBHEAD(screen));           /* Get DESK drop-down */
                                                /* and skip it        */
        for (; (drop = LWGET(OBNEXT(drop))) != screen; )
                {
                obj = LWGET(OBHEAD(drop));
                if (obj != NIL)
                if (dflt)
                        maptree(tree, obj, drop, enabobj);
                else
                        maptree(tree, obj, drop, disabobj);
                }

        for (; *change; change++)
                if (dflt)
                        disabobj(tree, *change);
                else
                        enabobj(tree, *change);
        }
```

```
/*
>>>>>>>>>>>>>>>>>>>> Definitions used in this article <<<<<<<<<<<<<<<<<<<<
*/

#define ROOT 0

#define GIBOX    25
#define GSTRING  28
#define GTITLE   32

#define RTREE     0

#define MNSELECTED 10

#define CHECKED   0x4
#define DISABLED  0x8

#define OBNEXT(x) (tree + (x) * sizeof(OBJECT) + 0)
#define OBHEAD(x) (tree + (x) * sizeof(OBJECT) + 2)
#define OBTAIL(x) (tree + (x) * sizeof(OBJECT) + 4)
#define OBTYPE(x) (tree + (x) * sizeof(OBJECT) + 6)
#define OBFLAGS(x) (tree + (x) * sizeof(OBJECT) + 8)
#define OBSTATE(x) (tree + (x) * sizeof(OBJECT) + 10)
#define OBSPEC(x) (tree + (x) * sizeof(OBJECT) + 12)
#define OBX(x) (tree + (x) * sizeof(OBJECT) + 16)
#define OBY(x) (tree + (x) * sizeof(OBJECT) + 18)
#define OBWIDTH(x) (tree + (x) * sizeof(OBJECT) + 20)
#define OBHEIGHT(x) (tree + (x) * sizeof(OBJECT) + 22)

#define MOFF      256
#define MON       257
```

Appendix - VII
Sample Code for Part IX

```
/*
>>>>>>>>>>>>>>>> Routines to set clip to a GRECT <<<<<<<<<<<<<<<<
*/

        VOID
grecttoarray(area, array)     /* convert x,y,w,h to upr lt x,y and    */
        GRECT   *area;          /*                      lwr rt x,y      */
        WORD    *array;
        {
        *array++ = area->gx;
        *array++ = area->gy;
        *array++ = area->gx + area->gw - 1;
        *array = area->gy + area->gh - 1;
        }

        VOID
setclip(clipflag, sarea)     /* set clip to specified area           */
        WORD    clipflag;
        GRECT   *sarea;
        {
        WORD    pxy[4];

        grecttoarray(sarea, pxy);
        vsclip(vdihandle, clipflag, pxy);
        }
```

```
/*
>>>>>>>>>> Routines to set attributes before output <<<<<<<<<<<
*/
        VOID
rrperim(mode, color, type, width, pxy)      /* Draw a rounded    */
        WORD    mode, color, width, *pxy;    /* rectangle outline */
        {
        vswrmode(vdihandle, mode);
        vslcolor(vdihandle, color);
        vsltype(vdihandle, type);
        vslwidth(vdihandle, width);
        vrbox(vdihandle, pxy);
        vswrmode(vdihandle, MDREPLACE);
        }

        VOID
plperim(mode, type, color, width, npts, pxy)     /* Draw a polygonal */
                                                  /* figure           */
        WORD    mode, type, color, width, npts, *pxy;
        {
        vswrmode(vdihandle, mode);
        vsltype(vdihandle, type);
        vslcolor(vdihandle, color);
        vslwidth(vdihandle, width);
        vpline(vdihandle, npts, pxy);
        }

        VOID                    /* Draw a filled polygonal area */
plfill(mode, perim, color, interior, style, npts, pxy)
        WORD    mode, perim, color, interior, style, npts, *pxy;
        {
        vswrmode(vdihandle, mode);
        vsfcolor(vdihandle, color);
        vsfstyle(vdihandle, style);
        vsfinterior(vdihandle, interior);
        vsfperimeter(vdihandle, perim);
        vfillarea(vdihandle, npts, pxy);
        }

        VOID                    /* Draw a filled rectangle    */
rectfill(mode, perim, color, interior, style, pxy)
        WORD    mode, perim, color, style, interior, *pxy;
        {
        vswrmode(vdihandle, mode);
        vsfcolor(vdihandle, color);
        vsfstyle(vdihandle, style);
        vsfinterior(vdihandle, interior);
        vsfperimeter(vdihandle, perim);
        vrrecfl(vdihandle, pxy);
        }
```

Appendix - VIII
Sample Code for Part X

```
/*
>>>>>>>>>>> Demonstration of byte alignment of window interior <<<<<<<<<<<
*/

#define FEATURES    0x0fef      /* what border features are used   */
WORD    msg[8];                 /* message from evntmulti          */
GRECT   workarea;               /* defines working area            */
WORD    whndl;                  /* handle for window being changed */

        windcalc(1, FEATURES, msg[4], msg[5], msg[6], msg[7],
                &workarea.gx, &workarea.gy, &workarea.gw,
                &workarea.gh);
        workarea.gx = alignx(workarea.gx);
        workarea.gw = alignx(workarea.gw);
        windcalc(0, FEATURES, workarea.gx, workarea.gy,
                workarea.gw, workarea.gh, &msg[4], &msg[5],
                &msg[6], &msg[7]);
        windset(whndl, WFCXYWH, msg[4], msg[5], msg[6], msg[7]);
```

```
/*
>>>>>>>>>>>>>>>>>>>> Subroutine for above <<<<<<<<<<<<<<<<<<<<<<<<<<<<<
*/
        WORD
alignx(x)               /* forces word alignment for column position    */
        WORD    x;              /*   rounding to nearest word           */
        {
        return((x & 0xfff0) + ((x & 0x0008) ? 0x0010 : 0));
        }
```

```
/*
>>>>>>>>>>>>>>>>>>>> Standard vgtext binding <<<<<<<<<<<<<<<<<<<<<<<<
*/
        WORD
vgtext( handle, x, y, string)
        WORD handle, x, y;
        BYTE *string;
        {
        WORD i;
        ptsin[0] = x;
        ptsin[1] = y;
        i = 0;
        while (intin[i++] = *string++)  /* Copy characters to intin    */
                ;                       /* There is NO error checking! */
        contrl[0] = 8;
        contrl[1] = 1;
        contrl[3] = --i;
        contrl[6] = handle;
        vdi();
        }
```

Table of Contents